¿Qué se va a desarrollar?

Vamos a desarrollar desde cero una cadena de bloques, utilizando el lenguaje de programación Python y principalmente el paradigma orientado a objetos.

La blockchain que desarrollemos necesita cumplir con los siguientes requisitos para considerarse un funcional:

- Incluya transacciones entre dos usuarios en la red.
- Trabaje con el consenso Proof of Work y Proof of Stake.
- Incluya cifrado asimétrico de un solo camino.

Para desarrollarlo primero necesitamos tener las siguientes clases en archivos distintos:

1. Transaction
2. Account
3. Block
4. Consesos
    - 4.1 Proof of Work
    - 4.2 Proof of Stake
5. Blockchain

Requerimientos

Lo primero es haber tomado nota de los conceptos vistos en los temas previos por escrito para entender con más facilidad el porqué se hacen las cosas.

Vamos a necesitar instalar una dependencia llamada pycryptodome. Gracias a ella podremos trabajar con cifrados asimétricos, hashes, firmas y otras funciones importantes

In [2]:
```
import sys
!{sys.executable} -m pip install --upgrade pip
!{sys.executable} -m pip install pycryptodome
```

```
Requirement already satisfied: pip in c:\users\brian\appdata\local\packages\python
softwarefoundation.python.3.9_qbz5n2kfra8p0\localcache\local-packages\python39\sit
e-packages (22.3)
Requirement already satisfied: pycryptodome in c:\users\brian\appdata\local\packag
es\pythonsoftwarefoundation.python.3.9_qbz5n2kfra8p0\localcache\local-packages\pyt
hon39\site-packages (3.14.1)
```

Con los requerimientos instalados, podemos pasar al primer capitulo: Bloques.

Clase Block

Un bloque es donde se guarda la información en blockchain y se cifra.

En nuestro directorio de trabajo vamos a crear un archivo llamado block.py. En él, vamos a crear una nueva clase llamada Block.

In [5]:
```
class Block:
    pass
```

Para crear un bloque siempre vamos a necesitar 3 variables iniciales:

- El hash del bloque anterior (Con él se hace la vinculación entre bloque).
- Una lista que contenga las transacciones pendientes.
- El número del bloque.

In [1]:
```
class Block:
    def __init__(self, preivious_hash: str,
                 list_of_transactions: list,
                 block_number: int):
        pass
```

Nuestra clase Block va a contar con los siguientes atributos:

- Número del bloque: Altura del bloque actual en la cadena.
- Hash del bloque anterior: Firma digital del bloque anterior en la cadena.
- Lista de transacciones: Transacciones contenidas y procesadas.
- Hash del bloque actual: Firma digital del bloque actual.
- Timestamp: Hora en la que el bloque se añadió a la cadena de bloques.
- Nonce: Número mágico que resuelve el "acertijo" en el consenso Proof of Work.

In [9]:
```
class Block:
    def __init__(self, previous_hash: str, list_of_transactions: list, block_number
        self.block_number = block_number
        self.previous_hash = previous_hash
        self.list_of_transactions = list_of_transactions
        self.nonce = 0
        self.hash = 0
        self.time_stamp = datetime.now().strftime("%d/%m/%Y %H:%M:%S")
        # Cuando se instancia un bloque, le asigna el blocknumber a cada transacci
        for tx in self.list_of_transactions:
            tx.block = block_number
        # atributos utilizados con Proof of Stake.
        self.forger = None
```

Los métodos van a darnos utilidad para la blockchain, o para efectos de visualización. La clase Block va a tener los siguientes métodos:

In [2]:
```
def get_block_header(self):
        """Funcion que retorna el encabezado del bloque."""
        return {
            'previous_block_hash':self.previous_hash,
            'nonce': self.nonce,
            'transactions':self.get_tx_in_format(),
        }
```

get_block_header regresa un diccionario que se utiliza para conseguir el hash de un bloque.

In [10]:
```
def get_block_header_pos(self):
        return {
            'previous_block_hash':self.previous_hash,
            'nonce': self.nonce,
            'transactions':self.get_tx_in_format(),
            'forger': self.forger.validator.account.identity
            }
```

get_block_header_pos regresa un diccionario añadiendo al forjador del bloque en el caso de un consenso Proof of Stake. Se utiliza tambien para obtener el hash de un bloque.

In [11]:
```
    def print_block_info(self):
        print("-------------")
        print("Bloque No: ", self.block_number)
        print("Transacciones: ")
        self.print_tx_in_format()
        print("Hash anterior: ", self.previous_hash)
        print("Hash actual: ", self.hash)
        print("Time stamp: ", self.time_stamp)

    def print_tx_in_format(self):
        for tx in self.list_of_transactions:
            print(
                f"- {tx.sender.nickname} send {tx.value} to {tx.recipient.nickname}

    def get_tx_in_format(self):
        tx_list = []
        for tx in self.list_of_transactions:
            tx_in_str = f"{tx.sender.nickname} send {tx.value} to {tx.recipient.nic
            tx_list.append(tx_in_str)
        return str(tx_list)
```

Las funciones anteriores solo imprimen información para visualizarla de mejor manera en la práctica.

El código completo queda de la siguiente manera:

In [4]:
```
from datetime import datetime, date
from bin.account import Account


class Block:
    def __init__(self, previous_hash: str, list_of_transactions: list, block_number
        self.block_number = block_number
        self.previous_hash = previous_hash
```

```
        self.list_of_transactions = list_of_transactions
        self.nonce = 0
        self.hash = 0
        self.time_stamp = datetime.now().strftime("%d/%m/%Y %H:%M:%S")
        # Cuando se instancia un bloque, le asigna el blocknumber a cada transacci
        for tx in self.list_of_transactions:
            tx.block = block_number
        # atributos utilizados con Proof of Stake.
        self.forger = None

    def get_block_header(self):
        """Funcion que retorna el encabezado del bloque."""
        return {
            'previous_block_hash':self.previous_hash,
            'nonce': self.nonce,
            'transactions':self.get_tx_in_format(),
        }

    def get_block_header_pos(self):
        return {
            'previous_block_hash':self.previous_hash,
            'nonce': self.nonce,
            'transactions':self.get_tx_in_format(),
            'forger': self.forger.validator.account.identity
            }

    def print_block_info(self):
        print("-------------")
        print("Bloque No: ", self.block_number)
        print("Transacciones: ")
        self.print_tx_in_format()
        print("Hash anterior: ", self.previous_hash)
        print("Hash actual: ", self.hash)
        print("Time stamp: ", self.time_stamp)

    def print_tx_in_format(self):
        for tx in self.list_of_transactions:
            print(
                f"- {tx.sender.nickname} send {tx.value} to {tx.recipient.nickname}

    def get_tx_in_format(self):
        tx_list = []
        for tx in self.list_of_transactions:
            tx_in_str = f"{tx.sender.nickname} send {tx.value} to {tx.recipient.nic
            tx_list.append(tx_in_str)
        return str(tx_list)
```

Con la clase Block lista, podemos pasar a la clase Account.

Clase Account

Una cuenta es el medio por el que un usuario puede interactuar en la blockchain, tener activos y hacer transacciones hacia otros usuarios.

Vamos a crear un archivo en nuestro directorio de trabajo llamado account.py. En él, vamos a instanciar una clase de nombre Account.

In [1]:
```
class Account():
    pass
```

Cada que se instancie una cuenta, vamos a inicializarla con una cantidad de balance a elección y nickname para efectos de práctica.

In [8]:
```
class Account():
    def __init__(self, balance: int, nickname: str):
        pass
```

Como atributos incluiremos un nickname, un balance de su "dinero" y un historial que registra las transacciones que han realizado.

In [1]:
```
class Account:
    def __init__(self, balance:int, nickname: str):
        self.nickname = nickname
        self.balance = 100
        self.list_of_all_transactions = []
```

En blockchain, y hablando específicamente en el contexto de la seguridad de una cuenta, cada cuenta tiene un cifrado asimétrico, teniendo llaves públicas y privadas.

Cada objeto Account tendrá una llave pública que funge como un identificador público para la cuenta, y una llave privada para autorizar transacciones de la cuenta.

Primero, vamos a importar dos funciones que vamos a usar del módulo Crypto:

- PublicKey.RSA
- Signature.pkcs1_15.PKCS115_SigScheme

In [18]:
```
from Crypto.PublicKey import RSA
from Crypto.Signature.pkcs1_15 import PKCS115_SigScheme

class Account:
    def __init__(self, balance: int, nickname: str):
        self.nickname = nickname
        self.balance = 100
        self.list_of_all_transactions = []
        # Cifrado asimetrico
        self.private_key = RSA.generate(1024) # Llave privada con algoritmo RSA de
        self.public_key = self.private_key.publickey() # Llave publica
        self.signer = PKCS115_SigScheme(self.private_key) # (1)
        self.verifier = PKCS115_SigScheme(self.public_key) # (2)
```

- La RSA nos ayuda a instanciar nuestra llave privada y la pública.
- PKCS115_SigScheme nos ayuda con un objeto de tipo "firmador". Con el, podemos firmar transacciones con nuestra llave privada, y verificar transacciones con nuestra llave publica.

Ejemplo visual.

Vamos a desarrollar un ejemplo visual para ver como se visualizan los objetos y como se ven.

Visualizacion de las llaves

In [22]:
```
from Crypto.PublicKey import RSA
private_key = RSA.generate(1024)
private_key.exportKey()
```

Out[22]:
```
b'-----BEGIN RSA PRIVATE KEY-----\nMIICWwIBAAKBgQDFIL3wCiQSg5OlR8NDmg2xzb8rbg7Tz8w
7c+pw2PCyLPbACbab\np6TXOc8jamUTWPiH2V2F16YVVel+q+mTuB70R7niQEYyNHYVoHiw6wxzLRgWCZX
y\ntml6zd20B5CRdFEkkX7qZk8zp5FopEjvHqsJGLyvCp0J0SpjbcZ5FZ0KEQIDAQAB\nAoGACUlvGVFo/
KRNPuKgHN2zmnZOdM3VDf6CqSRfXZyizkoW/9oVS8T8m5UiIJGB\nHIW7A69mXAlaTjQDcY/Iy8hHUPaRK
++oEkzbpYnfr0bPcjtSH9aPo6ga+UHrlbqs\nVbrhr45FRAv6JeGUY1Z/Nddsx7GS+hp1mGHQfCjWKkhXy
uECQQDLlagFQDU300/O\nz4vEpf9z+ZROg+hiaGJ+EpobiNetAXngtP9oJaRXexlOGfKyNz+umybe2Qwmk
Q3D\nHl2hEx4pAkEA9+GFp+58RGc09v7TRZlmWgcR8itaUiC9B/XqGaDLdoR9IeZKPfMD\ndTXy5V+rpzq
baKpzh8jlYTrwJutXZu85qQJAWkzWSwXw1QMwmg3q892hUkK4qp7N\nm6CrVzpPCrmG2KEX+zitNPfFTlw
2nDLcOHpTD9KXyi7BufWLAinc485ECQJAd+gZ\n1VAhwJ0EG+7MmEBKKIZ/AdxCxrObfTxRz6/efCg+t6V
EiI8DPzGnm5kZ2b0Z7Bl3\n5VkizFPMU6dIEjzqcQJAO+iIIUWmsQFx667W5m8OJHGeK6TG4g/VZ7fZzLC
q8est\nnSf/QYC6QnqUjhDtwkg5ZtOOMzPwPSRggMT59FiCNQ==\n-----END RSA PRIVATE KEY-----
'
```

In [64]:
```
private_key.publickey().exportKey() # LLave publica
```

Out[64]:
```
b'-----BEGIN PUBLIC KEY-----\nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC2fTuCQuNf2PWg
mP4Nncfu2hq0\nkEP62lqtm4HYvTiZjvRbmQ5X6gHU3QSpE+x0lVXDV4/kaZQBSO/gCMxWOmqvvaYh\nN+
2GmTH4be4ddyBN41QLXzOmtIGKtkQ69gxg/7GLzu4I4nil7k7cbc+lTCgDM/XI\nJyi/55twZjBwj3iyWw
IDAQAB\n-----END PUBLIC KEY-----'
```

Como vemos, es un bunche aleatorio de carácteres alfanumericos. Aunque...

La llave privada se suele visualizar de otra manera, esta manera la vamos a llamar 'identidad'.

Identidad

Las llaves públicas son el identificador de una cuenta, pero no se presentan o se visualizan como se ve arriba. Hay que convertirlo a hexadecimal para que sea visiblemente más accesible.

In [23]:
```
import binascii
binascii.hexlify(private_key.publickey().exportKey(format="DER")).decode('ascii') #
# este funcion convierte lo que exporta RSA en hexadecimal, haciendolo mas visible.
```

Out[23]:
```
'30819f300d06092a864886f70d010101050003818d0030818902818100c520bdf00a24128393a547c
3439a0db1cdbf2b6e0ed3cfcc3b73ea70d8f0b22cf6c009b69ba7a4d739cf236a651358f887d95d85d
7a61555e97eabe993b81ef447b9e2404632347615a078b0eb0c732d18160995f2b6697acdddb407909
1745124917eea664f33a79168a448ef1eab0918bcaf0a9d09d12a636dc679159d0a110203010001'
```

Signer y Verifier de una cuenta

Cada cuenta va a tener, digamos, dos ítems. Una va a ser una pluma, o una "firmadora", que nos ayudara a "firmar" las transacciones. La otra es una "verificadora", que se asegura que esa firma es legítima y que ni el contenido, ni la firma, ni el remitente, hayan cambiado.

Derivadas de estas llaves, se obtienen estos ítems. (1 y 2)

- Para crear el signer, se necesita la llave privada de la cuenta para autorizar/firmar las transacciones.
- Para crear el verifier, se necesita la llave pública de la cuenta para verificar el contenido, autor y firma digital de la transacción.

Con la biblioteca Crypto podemos usar el modulo Signature.pkcs1_15 para obtener un objeto de tipo PKCS115_SigScheme. Con él, podemos instanciar nuestra firmadora y nuestra verificadora.

In [27]:
```
firmadora = PKCS115_SigScheme(private_key)  # Se instancia con la llave privada
verificadora = PKCS115_SigScheme(private_key.publickey())  # Se instancia con la ll
```

In [28]:
```
firmadora, verificadora
```

Out[28]:
```
(<Crypto.Signature.pkcs1_15.PKCS115_SigScheme at 0x104682fb0>,
 <Crypto.Signature.pkcs1_15.PKCS115_SigScheme at 0x1046823e0>)
```

En la clase Account solo vamos a implementar un método. El codigo completo de la clase Account es el siguiente.

In [1]:
```
import binascii
from Crypto.PublicKey import RSA
from Crypto.Signature.pkcs1_15 import PKCS115_SigScheme

class Account:
    def __init__(self, balance: int, nickname: str):
        self.nickname = nickname
        self.balance = balance
        self.list_of_all_transactions = []
        self.private_key = RSA.generate(1024)
        self.public_key = self.private_key.publickey()
        self.signer = PKCS115_SigScheme(self.private_key)
        self.verifier = PKCS115_SigScheme(self.public_key)

    @property
    def identity(self):
        return binascii.hexlify(self.public_key.exportKey(format="DER")).decode('as
```

Con la clase Account lista, podemos pasar a la clase Transaction.

Clase Transaction

Una transacción en el contexto de blockchain es un cambio de estado en la red. Las transacciones de nuestra blockchain permiten que dos usuarios puedan transferirse dinero virtual.

Vamos a crear un archivo en nuestro directorio de trabajo llamado transaction.py. En él, vamos a instanciar una clase de nombre Transaction.

In [3]:
```
class Transaction():
    pass
```

Cada transacción tiene 3 elementos principales.

- ¿Quién manda la transaccion? (sender)
- ¿Quién la recibe? (recipient)
- ¿Cuánto dinero es? (value)

Cada que se instancie una transacción en local, la vamos a inicializar con sus elementos principales.

In [4]:
```
from bin.account import Account

class Transaction():
    def __init__(self, sender: Account, value: int, recipient: Account):
        pass
```

Además, añadiremos atributos "informativos" que sirven para guardar información importante de la transaccion.

- Timestamp: Hora en la que la transaccion fue añadida a la red.
- Block: Bloque al que pertenece la transaccion.
- Signature: Firma digital de la transaccion.
- Status: Estado actual de la transaccion.

Así como añadir a la propia transaccion al historial de transacciones de quien envía la transaccion. Aunque se rechace la transacción, quedara el registro guardado en su cuenta.

In [5]:
```
from bin.account import Account

class Transaction:
    def __init__(self, sender: Account, value: int, recipient: Account):
        self.sender = sender
        self.value = value
        self.recipient = recipient
        # Hora en la que la transaccion se instancia
        self.time = datetime.now().strftime("%d/%m/%Y %H:%M:%S")
        self.block = None
        self.signature = None
        self.status = TxStatus.PENDIENTE
```

```
        # Al instanciarse una tx, esta debe reflejarse en la cuenta que la envia. (
        sender.list_of_all_transactions.append(self)
```

El atributo status lo necesitamos para ir cambiando si nuestra transacción esta en espera, pendiente o aprobada. Para ello, vamos a crear una clase Enum que nos ayude a hacerlo visible.

Enum

Un Enum nos ayuda a definir nuestros propios tipos de datos. En este caso, vamos a diseñar uno que nos dé 3 casos de una transacción:

- Completada
- Rechazada
- En espera

La clase TxStatus la vamos a poner antes de nuestra clase Transaction en nuestro archivo transaction.py

In [6]:
```
from enum import Enum
from bin.account import Account

class TxStatus(Enum):
    PENDIENTE = 0
    CONFIRMADA = 1
    DECLINADA = 2

class Transaction():
    def __init__(self, sender: Account, value: int, recipient: Account):
        ...
        self.status = TxStatus.PENDIENTE
```

Un ejemplo visual de cómo se veria seria el siguiente:

In [7]:
```
tx = Transaction(None, None, None)
tx.status
```

Out[7]:
```
<TxStatus.PENDIENTE: 0>
```

Hablando de sus métodos...

Implementaremos un método llamado **to_dict** que exporte un diccionario con el "encabezado" de la transacción. Es la información que se codifica por un hash a la hora de obtener su firma digital.

In [8]:
```
    def to_dict(self):
        """Exporta la transaccion en formato: dict."""
        return {
            'sender': self.sender.nickname,
            'recipient': self.recipient.nickname,
            'value': self.value,
            'time' : self.time}
```

Implementaremos dos métodos que nos ayuden a:

- Firmar transacciones.
- Verificar la firma digital.

Pero para ello, primero veamos este ejemplo:

Firmado y Verificado de transacciones utilizando una cuenta

Firmado

Digamos que nuestra transacción es lo siguiente:

In [9]:
```
transaction = {
    "sender": "Pedro",
    "receiver": "maria",
    "amount": 10
}
```

Si trataramos de ingresar en un hash este contenido tendriamos un error muy grande: (probar en local)

In [1]:
```
from Crypto.Hash import SHA256
# hash = SHA256.new(transaction) # Sale un errorsaso.
```

El algoritmo SHA256 solo acepta valores tipo byte, por lo tanto necesitamos convertir nuestro diccionario en una cadena de bytes.

Primero convertimos de dict -> str y después de str -> bytes.

In [11]:
```
transaction_str = str(transaction)
# Ya lo convertimos a String, ahora necesitamos convertirlo a bites
transaction_byte = transaction_str.encode()
type(transaction_byte)
```

Out[11]:
```
bytes
```

Una vez lo tenemos en bytes podemos pasarlo por un algoritmo SHA256 sin problema.

In [12]:
```
tx_hashed = SHA256.new(transaction_byte)
# Imprimimos el hash en su version hexadecimal
tx_hashed.hexdigest()
```

Out[12]:
```
'6a9519ccc6c7ecececf8801e699fe72544eccb46a37a7387bc416d4f347141a8'
```

Ya que tenemos el contenido de nuestra transacción y su hash, podemos obtener su firma. Hagamos otro ejemplo.

Pedro es el dueño de la transacción. Por lo tanto, Pedro tiene una cuenta con sus respectivas llaves, y puede firmarlas con su llave privada.

In [13]:
```
from bin.account import Account
pedro = Account(100, 'pedro')

# Pedro procede a firmar la transacción
signature = pedro.signer.sign(tx_hashed)
```

```
# Nuestra firma se ve de la siguiente manera
signature
```

Out[13]:
```
b':\xf3V\n\x92SN\n\xf5\xd9\xc6Ym\xdc\xd7\xfb\xf2\xcb\xf9\xa2I\xdfI\xe5\xc7\xd5\x98
+E\xc8\xae\xa7<k\xb0SZ\x99/\xbe\xb4f\xa9\x9fS4\x9e\xe4\xda&H\x07Y\x9ez\x9c\xe2\xe3
\x85w\x08j)*.]\xb2\xed\x1ee\xde\x03\xc2x\x90r~\xde\x98D\xee:\xfe\xf8\x05\r\xad\x83
K7\xcd\x02\xca\xd5}\x8d\xfa4\xf9\xd3\xb5\x10\xe9T(\x9c\xc40{M\xc4\x13\xff\xa4\xae
\xfa\x04Q\x1b\xf2\xe8\xd5\xeb!\xc4p\x14\xcc'
```

Verificado

Para verificar una transacción el proceso es muy similar.

Primero hay que hacer el mismo proceso de pasar la transaccion a un algoritmo hash.

In [14]:
```
tx_hashed = SHA256.new(str(transaction).encode()) # Todo el hash en una sola linea
tx_hashed
```

Out[14]:
```
<Crypto.Hash.SHA256.SHA256Hash at 0x1d2a4ba3640>
```

Solo que en vez de usar el firmador, vamos a usar el verificador.

Para verificar necesitamos:

- La firma de la transacción.
- El hash de la transacción previamente obtenido.

El verificador alza un error si la firma es inválida. Por eso utilizamos try/except.

In [15]:
```
try:
    pedro.verifier.verify(tx_hashed, signature)
    print(True)
except:
    print(False)
```

```
True
```

In [16]:
```
# Si lo intentaramos con una firma falsa el resultado seria distinto
firma_falsa = SHA256.new('transaccion falsa'.encode())
try:
    pedro.verifier.verify(firma_falsa, signature)
    print(True)
except:
    print(False)
```

```
False
```

Las condiciones para que salga error son:

- El contenido de la transacción haya sido alterado.
- La firma es incorrecta.
- Quien firma no es el autor de la transacción.

Vamos a añadir estas funciones a los métodos de nuestra clase Transaction. Nuestro código completo quedaría así:

In [20]:
```
from datetime import datetime
```

```
from enum import Enum
from bin.account import Account
from Crypto.Signature.pkcs1_15 import PKCS115_SigScheme
from Crypto.Hash import SHA256

class TxStatus(Enum):
    PENDIENTE = 0
    CONFIRMADA = 1
    DECLINADA = 2

class Transaction:
    def __init__(self, sender: Account, value: int, recipient: Account):
        self.sender = sender
        self.value = value
        self.recipient = recipient
        self.time = datetime.now().strftime("%d/%m/%Y %H:%M:%S")
        self.block = None
        self.signature = None
        self.status = TxStatus.PENDIENTE
        # Al instanciarse una tx, esta debe reflejarse en la cuenta que la envia. (
        sender.list_of_all_transactions.append(self)


    def to_dict(self):
        """Exporta la transaccion en formato: dict."""
        return {
            'sender': self.sender.nickname,
            'recipient': self.recipient.nickname,
            'value': self.value,
            'time' : self.time}

    def sign_transaction(self): # (1)
        """Funcion que recibe un objeto transaccion y devuelve
        la firma de la transaccion en bytes"""
        print("Firmando transaccion...")
        msg = str(self.to_dict()).encode()
        hash = SHA256.new(msg)
        signer = self.sender.signer
        signature = signer.sign(hash)
        # print("Signature:", binascii.hexlify(signature))
        self.signature = signature

    def verify_signature(self) -> bool: # (1)
        """Aqui se verifican las transacciones"""
        print("Verificando la firma de la transaccion...")
        msg = str(self.to_dict()).encode()
        hash = SHA256.new(msg)
        verifier = self.sender.verifier
        try:
            verifier.verify(hash, self.signature)
            print("La firma es valida.")
            return True
        except:
            print("La firma es invalida.")
            return False

    def change_status(self, new_status): # (3)
        if new_status == 'CONFIRMADA':
            self.status = TxStatus.CONFIRMADA
        elif new_status == 'PENDIENTE':
            self.status = TxStatus.PENDIENTE
```

```
        elif new_status == 'DECLINADA':
            self.status = TxStatus.DECLINADA
```

Lo que aplicamos se encuentra en las funciones sign_transaction, verify_signature() y to_dict. Por otro lado...

Nuestro método change_status cambia el estado de una transacción. Hay que recordar que los estados cambian dependiendo del evento, por ejemplo:

- Cuando una transacción se crea, empieza siendo Pendiente.
- Cuando es parte de la cadena de bloques, pasa a estar confirmada.
- Si la red tiene algún error, podría rechazar la transacción.

Si ya vimos las cuentas, sus transacciones y el dónde se almacenan (bloques), solo nos queda ver el cómo se anexa un bloque a la red.

Proof of Work

Recordemos que Proof of Work, a grosso modo, son varios nodos utilizando su poder computacional para ganar una carrera. La carrera se gana resolviendo un "acertijo matemático" encontrando un número llamado "nonce". Los bloques pasan a ser parte de la red cuando el nonce es encontrado y los demás nodos llegan a un acuerdo.

Para este capítulo no haremos una clase. Nos vamos a centrar en acertijo que hay que resolver y el como se halla la solución.

El acertijo

Vamos a desarrollar un pequeño ejemplo para visualizar cómo se resuelve un acertijo en Proof of Work.

In [15]:
```
from Crypto.Hash import SHA256
contenido_a_cifrar = "ultra secreto"

# De manera normal, si hacemos un hash de lo de arriba se ve asi.
hash_normal = SHA256.new(contenido_a_cifrar.encode())
hash_normal.hexdigest()
```

Out[15]:
```
'ebd510e521801dc7ab91d89dcbbcd7aa30fd31ff8bb591401b2f8c87f1c2dd4a'
```

El acertijo en Blockchain suele ser encontrar un hash que cumpla ciertas condiciones:

1. Que los primeros "n" caracteres del hash sea 0. "n" podrían ser 3, 5 o 7 ceros.
2. Que el valor Hash en decimal, sea mayor a un target.

Veámoslo:

Si solo hacemos un hash no vamos a resolver nada. Necesitamos iterar muchas veces para encontrar un HASH que cumpla las condiciones. Para ello, existe el nonce.

Lo primero sería cifrar lo que sea que queramos cifrar.

In [1]:
```
from Crypto.Hash import SHA256

contenido_a_cifrar = {
    "contenido": "ultra secreto",
    "nonce": 0} # Incluimos el nonce en el contenido a cifrar
```

Definimos una dificultad "target" que nuestro hash debe de igualar o superar para considerar nuestro hash válido.

In [3]:
```
difficulty_hash = 0x0000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
difficulty_decimal = 176684706477838432958329750074291851582748389687561895812160
```

difficulty_hash y difficulty_decimal equivalen a lo mismo.

En este ejemplo, nuestro hash tiene que tener en sus primeros 4 digitos el número 0.

Obtenemos el primer hash con el nonce en 0.

In [4]:
```
hash_resultante = SHA256.new(str(contenido_a_cifrar).encode())
contenido_a_cifrar["nonce"], hash_resultante.hexdigest()
```

Out[4]:
```
(0, '63c03dc83f41c06e31d8daf87718110fe4b4dc8a6b47f2c9db8b1d2b52e7046f')
```

Para nada está cerca. Por lo que vamos a incluir un bucle While que vaya aumentando el nonce y revisando que el hash haya alcanzado el target.

In [5]:
```
# Solo Si el hash resultante es mayor o igual a el target de dificultad, puedes sal
# Si no cumple la condicion, se aumenta el valor del nonce, y se vuelve a cifrar el
while int(hash_resultante.hexdigest(), 16) >= difficulty_hash: # En vez de difficul
    contenido_a_cifrar["nonce"] += 1
    hash_resultante = SHA256.new(str(contenido_a_cifrar).encode())
print(hash_resultante.hexdigest(), ", nonce: ", contenido_a_cifrar["nonce"]) # El r
```

```
0000882567fe350c0c382ee3d5dd3bf34478069b8508a0e84f6124096949abae , nonce:  159043
```

Con el acertijo resuelto, un bloque está listo para ser anexado a la red.

El nonce se encuentra después de varias iteraciones. Todos los nodos que están compitiendo para ser los primeros en ganar la carrera buscan este valor, puede que algunos lo intenten con algún algoritmo distinto, pero en esencia es lo mismo.

En cadenas de bloques que contienen a muchos mineros en su red (Bitcoin), cuentan con un sistema en el que cada que se mina un bloque se ajusta la dificultad en relación a los mineros trabajando. Por ejemplo, Bitcoin ajusta su dificultad para que cada bloque se mine en aproximados 10 minutos.

La principal desventaja del consenso Proof of Work es el consumo excesivo de energía. Para resolver este problema, se creó un consenso más amigable para el ambiente llamado Proof of Stake. Veámoslo en el siguiente capítulo.

Proof of Stake

Proof of Stake renueva todo el sistema de "carrera de mineros" a "grupo de validadores sin confianza." Proof of Stake fue creado para solventar la principal problemática de Proof of Work (Gasto energético).

Ya no existen los mineros, ahora los nodos serán "validadores". En este consenso, "minar un bloque" pasa a ser "forjar un bloque", que significan lo mismo. Los Validadores pueden ejercer dos roles: Forjador y Testigo.

- El forjador se encarga de verificar las transacciones y agruparlas en un bloque, firmar ese bloque con su llave privada y enviarlo a la red.
- Los Testigos se encargan de revisar que el trabajo del forjador este correcto.
- Cuando el bloque haya sido validador. El forjador anexa el bloque a la Blockchain.

Antes, los mineros probaban su validez en la red con su trabajo. Ahora, prueban su validez con su "liquidez" (o stake). Cada nodo pone su dinero en la red para probar que es de confianza; si el nodo quisiera dañar la red, su dinero (que no es una cantidad pequeña) sería destruido.

En este consenso, entre más dinero tenga un nodo puesto como "seguro" en la red, más probabilidad tiene de ser el siguiente forjador del bloque.

Cada que se escoge un nuevo forjador, se escogen también un número determinado de validadores.

Nos vamos a centrar en todo este mecanismo, poniendo un ejemplo, para después ver como se implementa en una función de nuestra cadena de bloques.

Vamos a ir desarrollando poco a poco, clase por clase

Nodos

Un nodo que está participando para ser elegido como validador o como forjador, tiene que tener si o si una cuenta en la que recibiría las recompensas de la red. Una cuenta puede usar la red sin la necesidad de ser un validador. Un nodo puede ser un validador por dos motivos:

1. Porque el usuario así lo desea.
2. Porque cuenta con dinero suficiente para demostrar que es de confianza.

En Proof of Stake, se necesita un "mínimo" de dinero virtual para ser validador de la red.

Podemos concluir que: Si tienes una cuenta en la Blockchain, y tienes mínimo N cantidad de dinero virtual, puedes ser un validador.

Para la blockchain que vamos a desarrollar, vamos a crear una nueva clase llamada Validator en el mismo archivo en el que tenemos nuestra clase Account.

Esta clase va a tener dos atributos:

- "Tokens" por cuenta.
- La dirección de la cuenta que es Validador.

In [ ]:
```
from bin.tokens import Token
from bin.account import Account

class Validator():
    def __init__(self, account):
        self.account = account
        self.tokens = []

    def set_tokens(self, total_coins):
        """Funcion que por cada moneda en el balance, se instancia un nuevo token e
        for every_coin in range(0, total_coins):
            self.tokens.append(Token(self))

    def get_tokens(self):
        """Funcion que devuelve el numero de tokens en la cuenta."""
        return len(self.tokens)
```

Los **tokens** son objetos dentro de una blockchain que puede tener una utilidad dentro de la misma. En este caso, nos van a ayudar para resolver el algoritmo de selección de forjador dentro de un grupo de validadores. Vamos a crear una clase Token.

Tokens

In [ ]:
```
class Token():
    def __init__(self, owner):
        self.owner = owner
```

La clase Token va a tener una función muy sencilla, pero muy importante. Tener un dueño al cual nombrar ganador (esto se ve mas adelante).

¿Cómo se ve la clase Forjador y Testigo?

Clase Forger

Un forjador es encargado de crear un bloque y anexarlo a red. Funciona de la siguiente manera:

1. Primero agrupa las transacciones y las verifica.
2. Crea un bloque y anexa las transacciones.
3. Firma el bloque con su llave privada y lo envía a los validadores del bloque.
4. Los demás validadores ahora serán testigos, y su trabajo será atestiguar que el nodo hizo bien su trabajo.
5. Si todos llegan a un consenso, el forjador tendrá permiso para anexar el nodo a la red.

Dentro de nuestro directorio de trabajo vamos a crear un nuevo archivo python llamado forger.py.

En el vamos a instanciar una clase llamada Forger.

In [ ]:
```
class Forger():
    pass
```

Para crear a un forjador, se tiene que inicializar con el validador que ejercerá ese rol.

In [ ]:
```
class Forger():
    def __init__(self, validator):
        pass
```

La clase que vamos a usar en nuestra Blockchain va a necesitar los siguientes atributos:

- validator: Guarda la dirección de la cuenta que está realizando la transacción.
- block: Variable que almacena el bloque que se crea.
- block_signature: Variable que almacena la firma de bloque hecha por el forjador.

In [ ]:
```
class Forger():
    def __init__(self, validator):
        self.validator = validator # Cuando se crea un Forger, se vincula con la cu
        self.block = None # Bloque creado por el forjador
        self.block_signature = None # Firma del bloque
```

En cuanto a sus métodos...

Lo primero que hace un forjador es agrupar las transacciones en espera y verificarlas.

In [ ]:
```
    def verify_tx(self, holding_tx):
            """Funcion en la que el forjador verifica que las transacciones en espe
            validas para incluirlas en el bloque. La verificacion es igual que en P
            print('El forjador esta verificando las tx.')
            verified_tx = []
            for tx in holding_tx:
                if tx.verify_signature(): # Utiliza el metodo de la clase Transacti
                    print(tx.to_dict())
                    verified_tx.append(tx) # Si la firma se verifica, se anexa a ur
            return verified_tx
```

Una vez listas las transacciones, el forjador crea un objeto bloque.

In [ ]:
```
    def create_a_block(self, _previous_hash, _verified_tx, _block_number):
        self.block = Block(_previous_hash, _verified_tx, _block_number)
```

Con el bloque creado, y con las transacciones verificadas y almacenadas, el forjador firma el bloque. El proceso es similar a cuando se firma una transacción.

In [ ]:
```
    def sign_block(self) -> bool:
        """El forjador recibe un objeto bloque, lo firma con su llave privada."""
        print("### FIRMANDO EL BLOQUE")
        # Se añade el validador al bloque.
        self.block.forger = self # Antes de firmarse el bloque, se guarda en el blo
        block_header = json.dumps(self.block.get_block_header_pos()).encode() # Se
        block_hashed = SHA256.new(block_header) # El bloque pasa por un algoritmo S
        #
        signer = self.validator.account.signer # Cada forjador es una cuenta con el
        signature = signer.sign(block_hashed)
```

```
        self.block.hash = block_hashed.hexdigest() # Se almacena dentro del bloque
        self.block_signature = signature # El forjado guarda consigo su firma del b
        print('### BLOQUE FIRMADO Y CON SU HASH ---')

    def broadcast_block(self):
        """Funcion que retorna el bloque creado del forjador."""
        return self.block
```

Con la transacción firmada, se envía a la red.

In [ ]:
```
def broadcast_block(self):
    return self.block
```

Clase Attestor

Los Attestors (Testigos en ingles) son los encargados de verificar el trabajo del forjador; "atestiguan" que todo esté bien. Su metodología iría algo así:

- Reciben el bloque del forjador.
- Revisan que la firma del bloque sea correcta.
- Si no hay nada raro, y mas del 75% de los validadores estan de acuerdo, el bloque puede pasar a ser parte de la red.

Dentro de nuestro directorio de trabajo vamos a crear un nuevo archivo python llamado attestor.py.

En el vamos a instanciar una clase llamada Attestor.

In [ ]:
```
class Attestor():
    pass
```

Para crear a un testigo, se tiene que inicializar con el validador que ejercerá ese rol.

In [ ]:
```
class Attestor():
    def __init__(self, validator):
        pass
```

Como métodos solo vamos a necesitar una función que revise el bloque.

In [ ]:
```
    def check_block(self, block):
        """Funcion que recibe un bloque, lo verifica y retorna un valor booleano si
        # Asi como se hacian las verificaciones de transacciones, se vuelve a obter
        block_header = json.dumps(block.get_block_header_pos()).encode()
        block_hashed = SHA256.new(block_header)
        # recordemos que se usa el verificador quien forjo el bloque.
        verifier = block.forger.validator.account.verifier
        try:
            verifier.verify(block_hashed, block.forger.block_signature)
            print(f"{self.validator.account.nickname}: Confirmo que la firma del bl
            return True
        except:
            print("La firma es invalida.")
            return False
        #
```

En el archivo attestor.py también vamos a desarrollar una clase Attestors (en plural.). Esta clase va a tomar a los validadores que no ganaron el sorteo, y juntarlos a todos como testigos.

In [ ]:
```
class Attestor():
    ...

class Attestors():
    pass
```

Para crear un objeto de tipo Attestor, vamos a necesitar una lista de los validadores que pasarán a ser testigos.

In [ ]:
```
class Attestors():
    def __init__(self, validators: list):
        pass
```

Como atributos solo tendremos una variable list llamada group, encargada de contener a los validadores e instanciarlos como un clase de tipo Attestor.

In [ ]:
```
class Attestors():
    def __init__(self, validators):
        self.group = [Attestor(validator) for validator in validators]
```

Esta clase va a contar con un método llamado Attest, el cual pone a cada testigo a revisar el trabajo del forjador y juntar sus confirmaciones. Si la mayoría de testigos están de acuerdo, el bloque puede ser añadido a la red.

In [4]:
```
    def attest(self, block):
        """Funcion que recibe un bloque, y va pasando a cada attestor a revisar el
        confirmations = [] # Variable que almacena las confirmaciones de los testig
        for attestor in self.group: # Por cada testigos en el grupo de todos los te
            confirmations.append(attestor.check_block(block)) # Anexa su respuesta
        print('-----------------')
        # Se hace una division para cononcer cuantos votos minimos necesitamos, si
        minimun_votes = len(self.group) // 1.25
        if sum(confirmations) > minimun_votes: # Si las confirmaciones son mayores
            print('Confirmaciones suficientes para anadir el bloque.')
            return True
        else:
            print('Confirmaciones insuficientes para anadir el bloque.')
            return False
```

Algoritmo de selección de Forjador y Testigos.

La Blockchain que utilizan el consenso Proof of Stake tiene un algoritmo que escoge al forjador y sus testigos por diversos factores. En este ejemplo, vamos a desarrollar un algoritmo de tómbola que beneficie a los nodos con más tokens en stacking.

In [5]:
```
from bin.account import Account, Validator

validadores_del_bloque = {} # validadores del bloque confirmados
total_stacked = 0 # dinero que se almacena de los validadores
```

```
# Vamos a crear 5 cuentas. Cada una de ellas se va a instanciar como un validador.
# Si bien sabemos que el dinero no sale de la nada, ponerles un balance nos ayuda p
# efectos de práctica...

charles = Account(350, 'charles')
edwin = Account(500, 'edwin')
oliver = Account(200, 'oliver')
erick = Account(90, 'erick')
sonia = Account(275, 'sonia')

# Las juntamos en una variable
lista_cuentas = [charles, edwin, oliver, erick, sonia]

# Digamos que todos quieren ser validadores, solo van a pasar aquellos
# que tengan mas de 100 de balance en su cuenta.

# Lo primero es hacer ese filto
for cuenta in lista_cuentas: # Bucle for que recorre cada una de las cuentas
    if cuenta.balance >= 100: # Si tiene 100 o mas de balance puede ser validador
        # Instanciar un nuevo objeto Validator.
        new_validator = Validator(cuenta)

        # Variable que almacena el dinero que se va a intercambiar por tokens
        account_money = int(new_validator.account.balance)

        # A traves del objeto Validator, podemos acceder al objeto Account, y
        # despues al atributo balance para restarle lo que gasto en tokens.
        new_validator.account.balance -= account_money

        # Se utiliza al funcion set_tokens para instanciar un numero
        # determinado de tokens en la cuenta. Si el usuario tiene 200
        # de balance, se van a intercambiar por 200 tokens.
        new_validator.set_tokens(account_money)

        # Se almacena en un diccionario el validador y la cuenta.
        validadores_del_bloque.update({new_validator: account_money})

        # Se suma al total stackeado de la red el balance que se cambio por tokens
        total_stacked += account_money

# Al final vamos a tener una lista de validadores. (Menos Erick)
validadores_del_bloque, total_stacked
```

Out[5]:
```
({<bin.account.Validator at 0x106446620>: 350,
  <bin.account.Validator at 0x106445420>: 500,
  <bin.account.Validator at 0x10647cca0>: 200,
  <bin.account.Validator at 0x1064ece50>: 275},
 1325)
```

In [6]:
```
# Su balance tambien fue restado.
for keys in validadores_del_bloque.keys():
    print(f"{keys.account.nickname}, {keys.account.balance}")
```

```
charles, 0
edwin, 0
oliver, 0
sonia, 0
```

Una vez que tenemos la lista neta de los validadores que van a formar parte de la forja y validación del bloque, haremos el sorteo.

¿Cómo desarrollar el algoritmo?

Dentro de esta celda, se va a diseñar un algoritmo que escoja al siguiente forger, el siguiente ejemplo fue una implementación propia, pero cada blockchain puede variar. Veámoslo:

- Cada validador tiene una cantidad finita de tokens.
- Se van a ingresar los tokens de todos los validadores que van a participar en el sorteo en una sola lista.
- Cada token tiene un dueño, por lo tanto, se sabe de quién es.
- Cada ticket que ingresaron da una la posibilidad de ser el siguiente forjador del bloque.
- La lista se revuelve y se escoge un elemento al azar.
- El dueño del ticket ganador pasa a ser el forjador del nuevo bloque, y los no ganadores a ser los validadores.

In [7]:
```
from random import sample, choice
from bin.forger import Forger


# 1.- Lista que almacenara todos los tickets de la "rifa".
pool = []
# 2.- Bucle que recorrera a cada validador, y añadara sus tokens a la lista general
for validator in validadores_del_bloque.keys():
    pool += validator.tokens
# 3.- Se revuelve la lista. (Como si fuera un sorteo.)
print('Acumulando los tokens de los validadores en el servidor actual...')
print(len(pool), '- tokens acumulados.')
print('Revolviendo la lista...')
pool = sample(pool, len(pool))
print('Lista revuelta!')

# Bucle que valida que los tokens se hayan incluido bien
contador = 0
for validator in validadores_del_bloque.keys():  # Por cada validador, en la lista
    for token in pool: # Por cada ticken en el pool
        if token.owner.account.nickname == validator.account.nickname: # Si el toke
            contador += 1
    print(validator.account.nickname, contador) # Se imprimen los tickets de cada v
    contador = 0
# Fin de la validacion

ticket_winner = choice(pool)
forger = Forger(ticket_winner.owner)
print(f'El forjador del nuevo bloque sera... {forger.validator.account.nickname}')

# los validadores no ganadores del sorteo pasan a ser testigos.
# se remueve el forjador, asi solo quedan los testigos
validadores_sin_el_forjador = validadores_del_bloque.copy() # Se hace una copia de
validadores_sin_el_forjador.pop(forger.validator) # Se elimina al forjador de esta
"no ganadores",[validator.account.nickname for validator in validadores_sin_el_for
```

```
Acumulando los tokens de los validadores en el servidor actual...
1325 - tokens acumulados.
Revolviendo la lista...
Lista revuelta!
charles 350
edwin 500
oliver 200
sonia 275
El forjador del nuevo bloque sera... charles
```

Out[7]: ('no ganadores', ['edwin', 'oliver', 'sonia'])

Aquellos que no ganaron el sorteo, pasan a ser "testigos".

In [8]:
```
from bin.attestor import Attestor, Attestors

# Para instanciar al grupo de testigos, necesitamos usar la clase Attestors
attestors = Attestors(validadores_sin_el_forjador)
```

Ya tenemos a nuestro forjador y nuestro testigo, ¿Qué seguiria?

Procedimiento del Forjador y del Testigo

Lo primero que sucede es que el Forjador recibe un bloque y verifica sus transacciones.

In [19]:
```
from bin.account import Account
from bin.transaction import Transaction
from bin.block import Block

# Instanciamos un bloque previo al actual para efectos de práctica
cadena_de_bloques = [Block('0', [], 0)]

# Vamos a crear una transaccion entre dos cuentas.
tx = Transaction(Account(100, 'pedro'), 10, Account(100, 'maria'))

# Cada transaccion se firma con la llave privada del que envia la transaccion.
tx.sign_transaction()

# Se anade la tx a una lista de espera
holding_tx = [tx]

# El forjador verifica las transacciones en la lista
verified_tx = forger.verify_tx(holding_tx)

# El forjador crea un bloque
forger.create_a_block(cadena_de_bloques[-1].hash, verified_tx, len(cadena_de_bloque

# El forjador firma el bloque
forger.sign_block()

# El forjador envia el bloque a la red
bloque_firmado = forger.broadcast_block()

if attestors.attest(bloque_firmado):
    print('TODO CORRECTO, LLEGAMOS A UN ACUERDO')
    cadena_de_bloques.append(bloque_firmado)
cadena_de_bloques[1].block_number
```

```
Firmando transaccion...
El forjador esta verificando las tx.
Verificando la firma de la transaccion...
La firma es valida.
{'sender': 'pedro', 'recipient': 'maria', 'value': 10, 'time': '24/09/2022 17:20:2
0'}
### FIRMANDO EL BLOQUE
### BLOQUE FIRMADO Y CON SU HASH ---
edwin: Confirmo que la firma del bloque es correcta.
oliver: Confirmo que la firma del bloque es correcta.
sonia: Confirmo que la firma del bloque es correcta.
-----------------
Confirmaciones suficientes para anadir el bloque.
TODO CORRECTO, LLEGAMOS A UN ACUERDO
```

Out[19]: 1

Ya abarcamos todas las clases que se desarrollan dentro de la clase Blockchain. Llego la hora de ir a nuestra clase principal y ver cómo podemos integrar todo lo anterior.

Clase Blockchain

Vamos a desarrollar una blockchain que integre todos los capítulos anteriores.

Una blockchain es una cadena de bloques que se van anexando, y cada bloque se vincula con el bloque anterior. Los bloques incluyen transacciones (cambios de estado en la red). Las transacciones son hechas por los usuarios. La forma en la que se anexan los bloques a la red es por medio de un consenso (Proof of Work y Proof of stake)

Tenemos todo listo para hacer una Blockchain que integre todo lo que vimos con anterioridad.

Crearemos un archivo llamado blockchain.py en nuestro directorio de trabajo. En él, vamos a crear una nueva clase llamada Blockchain.

In [2]:
```
class Blockchain:
    pass
```

A diferencia de las clases anteriores, no vamos a necesitar mandar ninguna variable al instanciar un nuevo objeto Blockchain. Los ajustes los vamos a ir anexando por aparte.

Como atributos nuestra blockchain va a tener los siguientes atributos:

- chain: Lista que almacenara la cadena de bloques de la blockchain.
- tx_limit_per_block: Variable que define el límite de transacciones en espera antes de ser anexadas a la red.
- holding_tx: Lista que almacenara las transacciones en lista de espera.

Por parte del consenso Proof of Stake. Vamos a agregar atributos específicos.

- total_stacked: Número entero que define el dinero total en stack de toda la red.
- validators: Diccionario que almacena las direcciones de los validadores y su stack.
- last_block: Variable que almacena el próximo bloque a ser anexado a la red.

In [3]:
```
class Blockchain:
    def __init__(self):
        # Atributos
        self.protocol = None
        self.chain = []
        self.tx_limit_per_block = 1
        self.holding_tx = []
        # Atributos propios de PoS
        self.total_stacked = 0
        self.validators = {}
        self.last_block = None
```

Métodos

En esta sección vamos a explicar cada método y el porqué de su implementación.

Set Consensus

Necesitamos implementar un método que cambie por completo el sistema en el que nuestra blockchain funciona cambiando de consenso.

In [15]:
```
def set_consensus(self, protocol: str):
        if protocol not in ('PoW', 'PoS'):
            print('Seleccione un consenso de los dos disponibles.(PoW, PoS)')
            sys.exit()
        self.consensus = protocol
```

Genesis Block

Al momento de lanzar una blockchain a una red de prueba, lo primero es inicializarlo con un bloque génesis.

Cada bloque cuenta con su propio hash y el hash del bloque anterior, esto sirve para vincularse entre sí y darle inmutabilidad a la red; pero, ¿Qué pasa cuando hablamos del bloque 0? ¿Qué procede cuando el bloque 0 no tiene un bloque por detrás? ¿Cómo se vincula?

Hay que diseñar una función que nos ayude a resolver esta problemática.

In [11]:
```
def generate_genesis_block(self):
        """Inicializa el bloque genesis."""
        print('asdfas')
        if len(self.chain) == 0: # Comprueba que la blockchain este vacia.
            tx = Transaction(Account(0, "Genesis0"), 0, Account(0, "Genesis01"))
            block = Block('0', [tx], 0)
            self.mine(block)
            self.chain.append(block)
            block.list_of_transactions[0].change_status('CONFIRMADA')
        else:
            raise "Error: La Blockchain tiene que estar vacia."
```

Al comprobar que la blockchain esta vacía, puede procede a crear cuentas y transacciones que solo se usaran una vez para inicializar el bloque génesis.

¿Cómo recibe la blockchain las transacciones?

New Tx

Implementaremos un método que reciba una transacción hecha en local y la mande a todos los procesos que necesita para ser anexada a un bloque.

In [27]:
```
from account import Account

# Recibe los parametros de la transaccion.

def new_tx(self, _sender: Account, _value: int, _receiver: Account):
        """Recibe los parametros para instanciar un objeto de
        tipo Transaction; verifica que sea una transacción válida
        y lo añade a la holding list de la blockchain,
        para luego ser parte de un bloque. """
```

```
        # 0.5. Se debe verificar que quien manda la tx, tenga suficiente balance er
        if _value > _sender.balance:
            print()
            print('No tienes suficiente balance en tu cuenta.')
            return
        print('\n','Nueva transaccion detectada... Balance suficiente.')
        # 1. Instanciar un objeto transaccion.
        tx = Transaction(_sender, _value, _receiver)
        # 1.2 Al momento de instanciar el objeto, le restamos a la cuenta principal
        # el dinero que envio.
        _sender.balance -= _value
        print("Estado: {}".format(tx.status.name))
        # 2. Esta transaccion necesita ser firmada (confirmada).
        tx.sign_transaction()
        # 3. La firma pasa a ser verificada, con la finalidad de comprobar que sea
        # 3.1. Si es correcta signifa que puede agregarse a un bloque
        if tx.verify_signature() == True:
            if len(self.holding_tx) < self.tx_limit_per_block: # Revisa si aun cabe
                self.holding_tx.append(tx)
                print("Transaccion añadida a la espera.")
            if len(self.holding_tx) >= self.tx_limit_per_block: # Revisa si la list
                self.add_tx_to_block()
        else:
            return
```

¿Qué pasa cuando las tx en lista de espera estan listas para ser anexadas a un bloque?

Add tx to block

Nuestra blockchain va a tener un cierto número de transacciones límite, cuando el número de tx en lista de espera sea igual o mayor al límite, se llamara el método add_tx_to_block.

Este método se implementa dentro de la función new_tx.

Es el encargado de agrupar las transacciones en espera a un bloque, e implementar el bloque a la red. La preparación del bloque va a variar dependiendo del mecanismo de consenso actual de la blockchain.

Caso: Proof of Work

En el caso de ser Proof of Work, la metodología es la siguiente:

1. Instanciaremos un objeto Block, para ellos vamos a necesitar enviar:

- El hash del bloque anterior.
- La lista de transacciones que entraran al bloque.
- El número del bloque.

2. Enviar ese bloque a una función minado.

3. Minar el bloque para encontrar el nonce y un hash que cumpla con el target deseado.

4. Anexar el bloque a la cadena.

5. Vaciar la lista de transacciones en espera.
6. Agregar el block_number en las transacciones que ya forman parte de un bloque oficial en la cadena de bloques.
7. Cambiar el estatus de las transacciones dentro del último bloque y "enviar" el dinero a quien corresponda.

In [59]:
```
def add_tx_to_block(self):
        """Funcion que toma las transacciones en espera y procede a implementarlas
        para su posterior adision en la cadena de bloques."""
        print("### Creando nuevo bloque ###")
        print('### Bloque No. ', len(self.chain))
        _block_number = len(self.chain)
        # Consenso Proof of Work
        if self.consensus == 'PoW':
            print('En PoW')
            block = Block(previous_hash=self.chain[-1].hash,list_of_transactions=se
            self.mine(block) # Minar el bloque y hallar su nonce
            self.chain.append(block) # Anexar ese bloque a la red
            print("### Bloque creado. ###\n")
            self.holding_tx = [] # Vaciar la lista de transacciones pendiente
            for tx in block.list_of_transactions: # Bucle que asigna el block numbe
                tx.block = _block_number
            self.verify_latest_tx() # 6
            self.send_money_to_receivers()
```

Mine

Minar un bloque significa anexarlo a la red, pero primero hay que encontrar el nonce para vencer la dificultad. Implementaremos un método que reciba un bloque y lo pase por un proceso de "minado".

El proceso es idéntico a como lo vimos en el apartado de proof of work. Se obtiene el block header, se mina, se busca hash que venza al target, una vez encontrado se anexa el hash al bloque.

Este método se implementa dentro de la función add_tx_to_block.

In [5]:
```
def mine(self, block) -> None:
        """Funciona que mina el bloque.
        Funciona segun el protocolo de Proof of Stake. """
        print('Dentro de funcion minado...')
        # Primero obtenemos el string que contiene toda la informacion del bloque.
        block_header = json.dumps(block.get_block_header()).encode()
        block_hashed = SHA256.new(block_header)
        block_hash = block_hashed.hexdigest()
        # Proceso de minado
        if self.consensus == 'PoW':
        # 1. Dificultad
            difficulty_hash = 0x0000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
            difficult_decimal = 176684706477838432958329750074291851582748389687561
            # Sigue hasta que el hash sea menor o igual a la dificultad
            while int(block_hashed.hexdigest(), 16) >= difficulty_hash:
                block.nonce += 1 # Incremento del Nonce
                block_header = json.dumps(block.get_block_header()).encode() # Se c
```

```
                block_hashed = SHA256.new(block_header) # Pdd. Va a salir distinto
                print('Nonce Guess: ', block.nonce)
                print('Resultant Hash: ' + str(block_hashed.hexdigest()))
                print('Decimal value of hash: ' + str(int(block_hashed.hexdigest(),
                block_hash = block_hashed.hexdigest() # El bloque guarda el hash er
            print('Winner hash: ', block_hash)
            block.hash = block_hash
        if self.consensus == 'PoS':
            # En Proof of Stake vamos a saltar el proceso de minado.
            block.hash = block_hash
```

Verify latest tx

Función que cambia el status de una transacción a "confirmada".

Una transacción está confirmada cuando ya es parte de la cadena de bloques.

In [52]:
```
def verify_latest_tx(self):
        """Pone como confirmadas las transacciones que ya forman parte de la cadena
        Bloques original."""
        latest_block = self.chain[-1]
        block_transactions = latest_block.list_of_transactions
        for tx in block_transactions:
            tx.change_status('CONFIRMADA')
```

Send money to receivers

Función que envía el dinero, dependiendo del estatus de la transaccion.

- Si la transacción es confirmada, el dinero se envía al destinatario.
- Si la transacción es rechazada, el dinero se regresa al remitente.

En sí, en una blockchain el dinero no se envía. La blockchain cambia de "estado". Esta implementación funciona, pero no es exactamente la forma "verdadera".

In [6]:
```
def send_money_to_receivers(self):
        """Funcion que envía el dinero a los recipientes."""
        latest_block = self.chain[-1]
        block_transactions = latest_block.list_of_transactions
        for tx in block_transactions:
            # Condicional que cambia el estado de la transaccion a Declinada (Para
            # tx.change_status('DECLINADA')
            if tx.status.name == 'CONFIRMADA':
                tx.recipient.balance += tx.value
            if tx.status.name == 'DECLINADA':
                tx.sender.balance += tx.value
```

Caso: Proof of Stake

En un consenso Proof of Stake ya no hay mineros, hay validadores, que van interpretando dos roles, de Forjadores y Testigos, para anexar bloques a la red. Se desarrolla de la siguiente manera:

1. La red debe de tener validadores. Si no los tiene, se tienen que crear antes de hacer transacciones.

2. Llamar a la función select_the_forger() para seleccionar al nuevo forjador del bloque y a los no ganadores.
3. Los no ganadores del algoritmo de selección de forjador pasan a ser los nuevos testigos.
4. El forjador agrupa y verifica las transacciones que estén en espera.
5. El forjador crea un bloque (incluye las transacciones verificadas).
6. El forjador firma el bloque con su llave privada.
7. El forjador envía el bloque a la red.
8. Los testigos revisan que el trabajo que hizo el forjador sea válido. Si la mayor parte de los validadores están de acuerdo se llega a un acuerdo, y el bloque puede ser anexado a la red.
9. El forjador anexa el bloque a la red.
10. Se confirman las transacciones dentro del bloque y se "envía" el dinero a quien corresponda.

In [12]:
```
    def add_tx_to_block(self):
        """Funcion que toma las transacciones en espera y procede a implementarlas
        para su posterior adision en la cadena de bloques."""
        print("### Creando nuevo bloque ###")
        print('### Bloque No. ', len(self.chain))
        _block_number = len(self.chain)
        if self.consensus == 'PoS':
            print('En PoS:')
            forger, validators_with_out_forger = self.select_the_forger()
            # Los validadores no seleccionados pasan a ser objetos Attestors.
            attestors = Attestors(validators_with_out_forger)
            print('Testigos: ', [attestor.validator.account.nickname for attestor i
            print('Forjador: ', [forger.validator.account.nickname])
            # En proceso de verificar las tx...
            verified_tx = forger.verify_tx(self.holding_tx)
            forger.create_a_block(self.chain[-1].hash, verified_tx, _block_number)
            forger.sign_block()
            # el forjador manda el bloque a la red
            print('### Enviando el bloque a la red...')
            self.last_block = forger.broadcast_block()
            print('### Iniciando atestiguamiento del bloque...')
            if attestors.attest(self.last_block):
                self.chain.append(self.last_block)
                print("### Bloque creado. ###\n")
                self.holding_tx = []
                for tx in self.last_block.list_of_transactions:
                    tx.block = _block_number
                self.verify_latest_tx()
                self.send_money_to_receivers()
            else:
                print('No se pudo incluir el bloque.')
                return
```

1. La red debe de tener validadores. Si no los tiene, se tienen que crear antes de hacer transacciones.

Set Validators

Implementaremos un método que recibe una lista de cuentas interesadas en ser validadores y las instancíe como una clase Validator.

- Su balance es restado de sus cuentas, y guardado en la red.
- Se les otorga sus tokens correspondientes.
- Son agregados a los validores disponibles de la red.

In [138…
```
def set_validators(self, accounts):
        """Funcion que determina los validadores de la red."""
        for account in accounts:
            if account.balance >= 100: # Si tiene igual o mas de 100 en balance, pu
                new_validator = Validator(account) # Se crea un validador mandando
                self.total_stacked += new_validator.account.balance # Se anade el t
                new_validator.set_tokens(new_validator.account.balance) # Se anader
                self.validators.update({new_validator: new_validator.account.balanc
                new_validator.account.balance -= new_validator.account.balance # Se
```

2. Llamar a la función select_the_forger() para seleccionar al nuevo forjador del bloque y a los no ganadores.

Esta implementación ya la vimos en el capítulo de proof of stake. Recordemos que es una función qué implementa un algoritmo de tómbola, los participantes compran tickets, o en su caso Tokens, los Tokens entran a una lista, la lista se revuelve y se escoge al azar a un forjador, los no-ganadores pasan a ser los testigos. Se retorna el forjador y a los validadores no-ganadores.

In [139…
```
def select_the_forger(self):
        """Funcion que selecciona que validador va a ser el forjador del nuevo bloc
        # Dentro de esta funcion, se va a disenar un algoritmo que escoja al siguie
        #
        # Cada validador tiene un 'stack' dentro del total en la blockchain. Cada
        # moneda que ingresaron es un 'ticket' que les puede dar la posibilidad de
        # el siguiente forjador del bloque.
        # La cantidad de tickets va a depender de la cantidad de monedas que ingres

        # Este algoritmo escojera un ticket al azar. El dueno del ticker sera el nu
        # 1.- Lista que almacenara todos los tickets de la rifa
        pool = []
        # 2.- Bucle que recorrera a cada validador, y añadara sus tokens a la lista
        for validator in self.validators:
            pool += validator.tokens
        # 3.- Se revuelve la lista. (Como si fuera un sorteo.)
        print('Acumulando los tokens de los validadores en el servidor actual...')
        print(len(pool), '- tokens acumulados.')
        print('Revolviendo la lista...')
        pool = sample(pool, len(pool))
        print('Lista revuelta!')
        # Funcion que revisa los tokens de cada quien. Estaria chido ponerla como f
        contador = 0
        for validator in self.validators.keys():
            for token in pool:
                if token.owner.account.nickname == validator.account.nickname:
                    contador += 1
            print(validator.account.nickname, contador)
            contador = 0
        # Fin de la validacion
        # Aqui deberia de ir algo estilo, escojer el ticket ganador.
        ticket_winner = choice(pool)
        forger = Forger(ticket_winner.owner)
        print(f'El forjador del nuevo bloque sera... {forger.validator.account.nick
```

```
        # los validadores no ganadores del sorteo pasan a ser testigos.
        # los testigos estan encargados de revisar que el forjador haga lo correcto
        validators_with_out_forger = [validator for validator in self.validators]
        # se remueve el forjador, asi solo quedan los testigos
        validators_with_out_forger.remove(forger.validator)
        return forger, validators_with_out_forger
```

3. Los no ganadores del algoritmo de selección de forjador pasan a ser los nuevos testigos.

In [140…
```
def add_tx_to_block(self):
        if self.consensus == 'PoS':
            ...
            attestors = Attestors(validators_with_out_forger)
```

4. El forjador agrupa y verifica las transacciones que estén en espera.

Para este proceso el forjado utiliza el método verify_tx:

In [141…
```
def add_tx_to_block(self):
        if self.consensus == 'PoS':
            ...
            verified_tx = forger.verify_tx(self.holding_tx)
```

5. El forjador crea un bloque (incluye las transacciones verificadas).

In [142…
```
def add_tx_to_block(self):
        if self.consensus == 'PoS':
            ...
            forger.create_a_block(self.chain[-1].hash, verified_tx, _block_number)
```

6. El forjador firma el bloque con su llave privada.

In [143…
```
def add_tx_to_block(self):
        if self.consensus == 'PoS':
            ...
            forger.sign_block()
```

7. El forjador envía el bloque a la red.

In [144…
```
def add_tx_to_block(self):
        if self.consensus == 'PoS':
            ...
            self.last_block = forger.broadcast_block()
```

8. Los testigos revisan que el trabajo que hizo el forjador sea válido. Si la mayor parte de los validadores están de acuerdo se llega a un acuerdo, y el bloque puede ser anexado a la red.

In [145…
```
def add_tx_to_block(self):
        if self.consensus == 'PoS':
            ...
            if attestors.attest(self.last_block):
                print("Acuerdo concretado. El bloque puede anexarse a la red")

            else:
```

```
                print('No se pudo incluir el bloque.')
                return
```

9. El forjador anexa el bloque a la red.

In [146…

```
def add_tx_to_block(self):
        if self.consensus == 'PoS':
            ...
            if attestors.attest(self.last_block):
                print("Acuerdo concretado. El bloque puede anexarse a la red")
                self.chain.append(self.last_block)
                print("### Bloque creado. ###\n")
            else:
                print('No se pudo incluir el bloque.')
                return
```

10. Se confirman las transacciones dentro del bloque y se "envía" el dinero a quien corresponda.

In [147…

```
def add_tx_to_block(self):
        if self.consensus == 'PoS':
            ...
            if attestors.attest(self.last_block):
                print("Acuerdo concretado. El bloque puede anexarse a la red")
                self.chain.append(self.last_block)
                print("### Bloque creado. ###\n")

                # Se vacia la lista de tx pendientes
                self.holding_tx = []

                # se agrega el numero del bloque a la transacicon
                for tx in self.last_block.list_of_transactions:
                    tx.block = _block_number

                # se verifican las transacciones dentro del bloque
                self.verify_latest_tx()

                # el dinero se envia dependiendo del estado de las transacciones
                self.send_money_to_receivers()
            else:
                print('No se pudo incluir el bloque.')
                return
```

Esto seria todo de la clase Blockchain. Pasemos al siguiente capítulo para poner en práctica todo lo que hemos visto.

Ejemplo práctico

Caso: Proof of Work

Importamos la clase Account y Blockchain

In [2]:
```
from bin.account import Account
from bin.blockchain import Blockchain
```

Inicializar nuestra blockchain conlleva dos cosas:

1. Escoger el consenso con el que vamos a trabajar.
2. Inicializar el bloque génesis.

In [3]:
```
# Inicializamos nuestra cadena de bloques
print("### Inicializamos nuestra blockchain")
blockchain = Blockchain()  # Tambien se crea el bloque genesis.

# Escojemos que protocolo queremos en nuestra blockchain
blockchain.set_consensus('PoW')

# Inicializamos el bloque genes
blockchain.generate_genesis_block()
```

```
### Inicializamos nuestra blockchain
Inicializando bloque genesis...
Dentro de funcion mine...
Nonce Guess:  46378
Resultant Hash: 00001e6acd1d86b401d687712d322ab80a5af8f8f74214d7575cf629c52e135e
Decimal value of hash: 209931745916594597554099974976852832762162364941953807153725092211331934
```

Imprimimos el bloque génesis para revisar su creación.

In [4]:
```
blockchain.chain[0].print_block_info()
```

```
-------------
Bloque No:  0
Transacciones:
- Genesis0 send 0 to Genesis01
Hash anterior:  0
Hash actual:  00001e6acd1d86b401d687712d322ab80a5af8f8f74214d7575cf629c52e135e
Time stamp:  25/09/2022 17:21:29
```

Instanciaremos dos objetos Account que van a interactuar con la blockchain.

In [6]:
```
brian = Account(100, "Brian")
aaron = Account(100, "Aaron")
```

Con dos cuentas, podemos empezar a generar transacciones.

In [7]:
```
# Cada cuenta se instancio con 100 de balance, si nos pasamos, la transaccion no pa
blockchain.new_tx(brian, 120, aaron)
```

```
No tienes suficiente balance en tu cuenta.
```

In [8]:
```
# Cada cuenta se instancio con 100 de balance, si nos pasamos, la transaccion no pa
blockchain.new_tx(brian, 20, aaron)
```

```
Nueva transaccion detectada... Balance suficiente.
Estado: PENDIENTE
Firmando transaccion...
Verificando la firma de la transaccion...
La firma es valida.
Transaccion añadida a la espera.
### Creando nuevo bloque ###
### Bloque No.  1
En PoW
Dentro de funcion mine...
Nonce Guess:  175806
Resultant Hash: 0000972ca1031af0bbaff6eb93dab0aaaece8ee6923984d704ec489ce7566d55
Decimal value of hash: 104336689257510831139951500388979817668974780215209835373731169186704725

### Bloque creado. ###
```

La transaccion pasa por un mar de funciones dentro de la red.

1. La transacción se verifica.
2. Se añade a la espera.
3. Un nuevo bloque se mina y se adjuntan las transacciones en espera.
4. Se anexa a la red.

Observemos los cambios en la red.

In [52]:
```
# ¿Sí se transfierieron los activos?
brian.balance, aaron.balance
```

Out[52]:
```
(80, 120)
```

In [11]:
```
# Status de la transacción
last_transaction = brian.list_of_all_transactions[0]
last_transaction.status
```

Out[11]:
```
<TxStatus.CONFIRMADA: 1>
```

In [53]:
```
# Información de la transacción
last_transaction.to_dict()
```

Out[53]:
```
{'sender': 'Brian',
 'recipient': 'Aaron',
 'value': 20,
 'time': '25/09/2022 17:23:44'}
```

In [55]:
```
# También podemos acceder a la transacción desde los bloques
blockchain.chain[last_transaction.block].list_of_transactions[0].to_dict()
```

Out[55]:
```
{'sender': 'Brian',
 'recipient': 'Aaron',
 'value': 20,
 'time': '25/09/2022 17:23:44'}
```

Veamos la información del bloque a detalle.

In [58]:
```
blockchain.chain[-1].print_block_info()
```

```
-------------
Bloque No:  1
Transacciones:
- Brian send 20 to Aaron
Hash anterior:  00001e6acd1d86b401d687712d322ab80a5af8f8f74214d7575cf629c52e135e
Hash actual:  0000972ca1031af0bbaff6eb93dab0aaaece8ee6923984d704ec489ce7566d55
Time stamp:  25/09/2022 17:23:44
```

También podemos ver las firmas digitales hechas por las cuentas. Asi como, sus llaves, balance, etc.

In [57]:
```
blockchain.chain[-1].list_of_transactions[0].signature
```

Out[57]:
```
b'YqI\x83\xe2\x13\x94Z~\xa9\x0e2\xa14\x7f\xcb\x0f\xb6{\xea%\x07\xbb\xac\xa4\x17\x1
c\xca\xdfQ3y\xca\xd0\xe3g\x03\x0f\xe3\xf6\xa8\x81=\x0bV)\xd4\xa7FIe!\xf4W\x90\xa6
\x9d1\x9b\x90~\xfb\xd3\xd5/\x9b|\x95b\xe4\x86\x95\x8c\x88\x19\xdc\xeb\xa2\x80\xca
\xc4K\x94j\xce#\x7fh\x04\x81EKJ$lXX3g\xc9%\x16\xd5\nk\x99\xc4}\xec\x81\xc6\xf5\t]>
\x15\xa8jSc\xfe\xe0\x0e\x89\xb4\xb9\xaaz'
```

Es posible tambien ajustar el numero de transacciones que mantiene la blockchain en espera antes de ser ejecutadas.

In [59]:
```
blockchain.tx_limit_per_block = 3
```

In [60]:
```
blockchain.new_tx(aaron, 10, brian)
blockchain.new_tx(aaron, 30, brian)
blockchain.new_tx(brian, 10, aaron)
```

```
Nueva transaccion detectada... Balance suficiente.
Estado: PENDIENTE
Firmando transaccion...
Verificando la firma de la transaccion...
La firma es valida.
Transaccion añadida a la espera.
Nueva transaccion detectada... Balance suficiente.
Estado: PENDIENTE
Firmando transaccion...
Verificando la firma de la transaccion...
La firma es valida.
Transaccion añadida a la espera.
Nueva transaccion detectada... Balance suficiente.
Estado: PENDIENTE
Firmando transaccion...
Verificando la firma de la transaccion...
La firma es valida.
Transaccion añadida a la espera.
### Creando nuevo bloque ###
### Bloque No.  2
En PoW
Dentro de funcion mine...
Nonce Guess:  66743
Resultant Hash: 0000deebda7dd1c06da7f1c5390cf966403d77a9c04e78c7fa20b02bea8f45c4
Decimal value of hash: 153854628629294466645807724817771195071070517840276584062997
71981844825540

### Bloque creado. ###
```

In [61]:
```
blockchain.chain[-1].print_block_info()
```

```
-------------
Bloque No:  2
Transacciones:
- Aaron send 10 to Brian
- Aaron send 30 to Brian
- Brian send 10 to Aaron
Hash anterior:  0000972ca1031af0bbaff6eb93dab0aaaece8ee6923984d704ec489ce7566d55
Hash actual:  0000deebda7dd1c06da7f1c5390cf966403d77a9c04e78c7fa20b02bea8f45c4
Time stamp:  25/09/2022 17:38:04
```

Proof of stake

Igual que en Proof of Work, hay que inicializar nuestra blockchain.

El siguiente comando lo podemos utilizar para resetear las variables locales en Jupyter.

In [66]:
```
%reset
```

In [68]:
```
from bin.blockchain import Blockchain
from bin.account import Account, Validator
blockchain = Blockchain()

# Escojemos que protocolo queremos en nuestra blockchain
blockchain.set_consensus('PoS')

# inicializamos el bloque genesis
```

```
blockchain.generate_genesis_block()

blockchain.chain[0].print_block_info()
```

```
Inicializando bloque genesis...
Dentro de funcion mine...
Hash añadido al bloque genesis...
-------------
Bloque No:  0
Transacciones:
- Genesis0 send 0 to Genesis01
Hash anterior:  0
Hash actual:  7b36a0972430bcd8ee49cf3525d4ed88a7ce83e3c20097501fd086c7be62d3a5
Time stamp:  25/09/2022 17:39:44
```

In [69]:
```
# Creamos dos cuenta que interactuaran con la blockchain.
brian = Account(300, "Brian")
aaron = Account(300, "Aaron")
```

In [70]:
```
# Inicializamos 5 cuentas que quieran ser validadores
charles = Account(350, 'charles')
edwin = Account(500, 'edwin')
oliver = Account(200, 'oliver')
erick = Account(90, 'erick')
sonia = Account(275, 'sonia')
```

In [71]:
```
# Los incluimos en nuestra BlockChain
blockchain.set_validators((charles, edwin, oliver, erick, sonia))
```

In [72]:
```
# generamos y subimos las transacciones a la blockchain
blockchain.new_tx(brian, 50, aaron)
```

```
Nueva transaccion detectada... Balance suficiente.
Estado: PENDIENTE
Firmando transaccion...
Verificando la firma de la transaccion...
La firma es valida.
Transaccion añadida a la espera.
### Creando nuevo bloque ###
### Bloque No.  1
En PoS:
Acumulando los tokens de los validadores en el servidor actual...
1325 - tokens acumulados.
Revolviendo la lista...
Lista revuelta!
charles 350
edwin 500
oliver 200
sonia 275
El forjador del nuevo bloque sera... edwin
Testigos:  ['charles', 'oliver', 'sonia']
Forjador:  ['edwin']
El forjador esta verificando las tx.
Verificando la firma de la transaccion...
La firma es valida.
{'sender': 'Brian', 'recipient': 'Aaron', 'value': 50, 'time': '25/09/2022 17:42:5
8'}
### FIRMANDO EL BLOQUE
### BLOQUE FIRMADO Y CON SU HASH ---
### Enviando el bloque a la red...
### Iniciando atestiguamiento del bloque...
charles: Confirmo que la firma del bloque es correcta.
oliver: Confirmo que la firma del bloque es correcta.
sonia: Confirmo que la firma del bloque es correcta.
-----------------
Confirmaciones suficientes para anadir el bloque.
### Bloque creado. ###
```

La transacción pasa por aun más mar de transacciones.

1. Se verifica la firma de la transacción.
2. La transacción se añade a la espera.
3. Si no hay validadores, se crean.
4. Con validadores, se selecciona al forjador y a los testigos.
5. El forjador crea un bloque y adjunta las transacciones verificándolas. Lo firma con su llave privada y lo envía a los testigos.
6. Los testigos lo reciben, y revisan que el trabajo realizado por el forjador este bien.
7. Si la mayoría esta de acuerdo, si llega a una acuerdo y el bloque pasa a ser anexado a la red.
8. Se confirman las transacciones en la red.

Vamos a explorar un poco lo que obtuvimos.

In [73]:
```
# Si se transfieren los activos?
brian.balance, aaron.balance
```

Out[73]:
```
(250, 350)
```

In [74]:
```
# Status de la transaccion
last_transaction = brian.list_of_all_transactions[0]
last_transaction.status
```

Out[74]:
```
<TxStatus.CONFIRMADA: 1>
```

In [75]:
```
# Informacion de la transaccion
last_transaction.to_dict()
```

Out[75]:
```
{'sender': 'Brian',
 'recipient': 'Aaron',
 'value': 50,
 'time': '25/09/2022 17:42:58'}
```

Veamos la informacion del bloque a detalle.

In [77]:
```
blockchain.chain[-1].print_block_info()
```

```
-------------
Bloque No:  1
Transacciones:
- Brian send 50 to Aaron
Hash anterior:  7b36a0972430bcd8ee49cf3525d4ed88a7ce83e3c20097501fd086c7be62d3a5
Hash actual:  ecf72c08ef1ca765315abc4100cba8fa13875bf67713d5a02612489a9f7edbda
Time stamp:  25/09/2022 17:42:58
```

Tambien podemos ver las firmas digitales hechas por las cuentas. Asi como, sus llaves, balance, etc.

In [78]:
```
blockchain.chain[-1].list_of_transactions[0].signature
```

Out[78]:
```
b'\xc1i\n\xfd%y\xccojZ\xb2\x983`;\xad\xf6Z\xa0\x9aZX\x86[\x85\xea\xde\x075\xb3"\xd
3\x0b\x82\xfcr6\xd9c+\r\x13KF\x9c\x9aO\x9b\xc4\xc3,\xe5>\x1a)\xdb#\x15\xa4\x8c\x01
\r5\xaf\xd9\xb5\xbe\x0e\x92\x81\xc5\xec\x941\xe1\x9eD\x0e"\xec%>\xd0\xc6]\'\xa6\xf
1o\x03\xec\rl\xdb\x9fxU\xe2\x04\x10\xcab\xd5z\x12\x00\xb0\x08m\x0c\xa0\xbb\xb3>[z3
\xd6\xb3\\\x86iN\xb7\xdf7C\x84'
```

In [79]:
```
blockchain.chain[-1].print_block_info()
```

```
-------------
Bloque No:  1
Transacciones:
- Brian send 50 to Aaron
Hash anterior:  7b36a0972430bcd8ee49cf3525d4ed88a7ce83e3c20097501fd086c7be62d3a5
Hash actual:  ecf72c08ef1ca765315abc4100cba8fa13875bf67713d5a02612489a9f7edbda
Time stamp:  25/09/2022 17:42:58
```

En este caso, podemos observar quien fue el forjador del bloque.

In [80]:
```
blockchain.chain[-1].forger.validator.account.nickname
```

Out[80]:
```
'edwin'
```

In [88]:
```
blockchain.chain[-1].forger.block_signature
```

Out[88]:
```
b'\xad\x18\xd1\xab"8\x9cn\x13\xcdK\x06?kW=P>\xc3\xb7\xd0yG\xbf$\xabep\xad6\x88\xcf
rLf\xe9t\x171\xbc\xea?\x9b\xae\xb8E\xcb\xc5\x14\x0b\xa3\x90Xak\xf9PU\x8d\x98\x9b\x
d7\xd8\xed\x9c#&\xaa\xb3|\xfb\x87\x11\xed\x89\xaeV5\r\x1d\xb2Dy1\x8e\xd0Q\\\x7f\xc
0\xb6\x8a{\xfe\xea\x8e\x83G4:\x01\x11\x92-&o\xfbC{t\xfbp\xa1\xcb\xdd\x0e.\xd8\xfc%
\xd8\xb9\x8ac\xa0V1u'
```

Es posible tambien ajustar el numero de transacciones que mantiene la blockchain en espera antes de ser ejecutadas.

In [89]:
```
blockchain.tx_limit_per_block = 3
```

In [90]:
```
blockchain.new_tx(aaron, 10, brian)
blockchain.new_tx(aaron, 30, brian)
blockchain.new_tx(brian, 10, aaron)
```

```
Nueva transaccion detectada... Balance suficiente.
Estado: PENDIENTE
Firmando transaccion...
Verificando la firma de la transaccion...
La firma es valida.
Transaccion añadida a la espera.
Nueva transaccion detectada... Balance suficiente.
Estado: PENDIENTE
Firmando transaccion...
Verificando la firma de la transaccion...
La firma es valida.
Transaccion añadida a la espera.
Nueva transaccion detectada... Balance suficiente.
Estado: PENDIENTE
Firmando transaccion...
Verificando la firma de la transaccion...
La firma es valida.
Transaccion añadida a la espera.
### Creando nuevo bloque ###
### Bloque No.  2
En PoS:
Acumulando los tokens de los validadores en el servidor actual...
1325 - tokens acumulados.
Revolviendo la lista...
Lista revuelta!
charles 350
edwin 500
oliver 200
sonia 275
El forjador del nuevo bloque sera... sonia
Testigos:  ['charles', 'edwin', 'oliver']
Forjador:  ['sonia']
El forjador esta verificando las tx.
Verificando la firma de la transaccion...
La firma es valida.
{'sender': 'Aaron', 'recipient': 'Brian', 'value': 10, 'time': '25/09/2022 17:45:2
3'}
Verificando la firma de la transaccion...
La firma es valida.
{'sender': 'Aaron', 'recipient': 'Brian', 'value': 30, 'time': '25/09/2022 17:45:2
3'}
Verificando la firma de la transaccion...
La firma es valida.
{'sender': 'Brian', 'recipient': 'Aaron', 'value': 10, 'time': '25/09/2022 17:45:2
3'}
### FIRMANDO EL BLOQUE
### BLOQUE FIRMADO Y CON SU HASH ---
### Enviando el bloque a la red...
### Iniciando atestiguamiento del bloque...
charles: Confirmo que la firma del bloque es correcta.
edwin: Confirmo que la firma del bloque es correcta.
oliver: Confirmo que la firma del bloque es correcta.
-----------------
Confirmaciones suficientes para anadir el bloque.
### Bloque creado. ###
```

In [91]:
```
blockchain.print_full_chain()
```

```
-------------
Bloque No:  0
Transacciones:
- Genesis0 send 0 to Genesis01
Hash anterior:  0
Hash actual:  7b36a0972430bcd8ee49cf3525d4ed88a7ce83e3c20097501fd086c7be62d3a5
Time stamp:  25/09/2022 17:39:44
-------------
Bloque No:  1
Transacciones:
- Brian send 50 to Aaron
Hash anterior:  7b36a0972430bcd8ee49cf3525d4ed88a7ce83e3c20097501fd086c7be62d3a5
Hash actual:  ecf72c08ef1ca765315abc4100cba8fa13875bf67713d5a02612489a9f7edbda
Time stamp:  25/09/2022 17:42:58
-------------
Bloque No:  2
Transacciones:
- Aaron send 10 to Brian
- Aaron send 30 to Brian
- Brian send 10 to Aaron
Hash anterior:  ecf72c08ef1ca765315abc4100cba8fa13875bf67713d5a02612489a9f7edbda
Hash actual:  4e10a081c6594eed51c82bd49d71a47fb4343d180d6854d27d9943adbf7cb3e6
Time stamp:  25/09/2022 17:45:23
```

Llegamos al final de los ejercicios!

Conclusion

¡Muchas gracias por leer mi implementación de una blockchain! Ha sido un camino genial soltar el código y crear algo que pudiera enseñar los conceptos claves de una blockchain.

¡Doy gracias por la vida y a quienes me apoyaron a crear este contenido!

Espero esta información te haya ayudado a plantar mejor tus bases en blockchain y en programación. El trabajo está acabado, pero hay mucho más que se puede implementar.

"Algo perfecto es algo acabado, y en el crear, la perfección no existe".