Funzionamento della Blockchain con implementazione in Python

DI MASSIMO VAILATI
25/11/2025

PYTHON BLOCKCHAIN PROOF-OF-WORK HASH BITCOIN

SCIENTIFICO S.A. , TT INFORMATICA

La **blockchain** è un registro digitale distribuito, nato per garantire l'autenticità dei dati senza bisogno di un'autorità centrale. Ogni blocco contiene informazioni, un codice univoco (*hash*) e il riferimento al blocco precedente, formando una catena sicura e quasi immutabile. La validazione dei blocchi avviene tramite la **Proof of Work**, un processo che richiede tempo ed energia per evitare manomissioni. Oggi questa tecnologia è alla base di **Bitcoin**, ma trova applicazione anche in ambiti come sanità, arte digitale e contratti intelligenti, offrendo trasparenza e affidabilità nei sistemi informatici.

Autori

MASSIMO VAILATI

COS'È LA BLOCKCHAIN E COME FUNZIONA

La **blockchain** è una tecnologia ideata nel 1991 da due ricercatori americani. All'inizio serviva per garantire l'autenticità dei documenti digitali, evitando che potessero essere retrodatati o modificati di nascosto. In pratica, una specie di **notaio digitale**. Per molti anni però rimase inutilizzata, finché nel 2008 **Satoshi Nakamoto** la sfruttò per creare la prima criptovaluta della storia: il **Bitcoin**.

Immagina la blockchain come un a**rchivio digitale condiviso**, visibile e consultabile da tutti i partecipanti alla rete. La sua caratteristica principale è che, una volta registrate, le informazioni diventano **molto difficili da modificare**.
Il nome stesso ce lo spiega: **blockchain** significa **catena di blocchi**.
Ogni blocco contiene tre elementi fondamentali:

1. **I dati**: le informazioni che devono essere registrate.

   Ad esempio, nella blockchain di Bitcoin i dati riguardano una transazione: chi invia, chi riceve e quanti bitcoin vengono trasferiti.

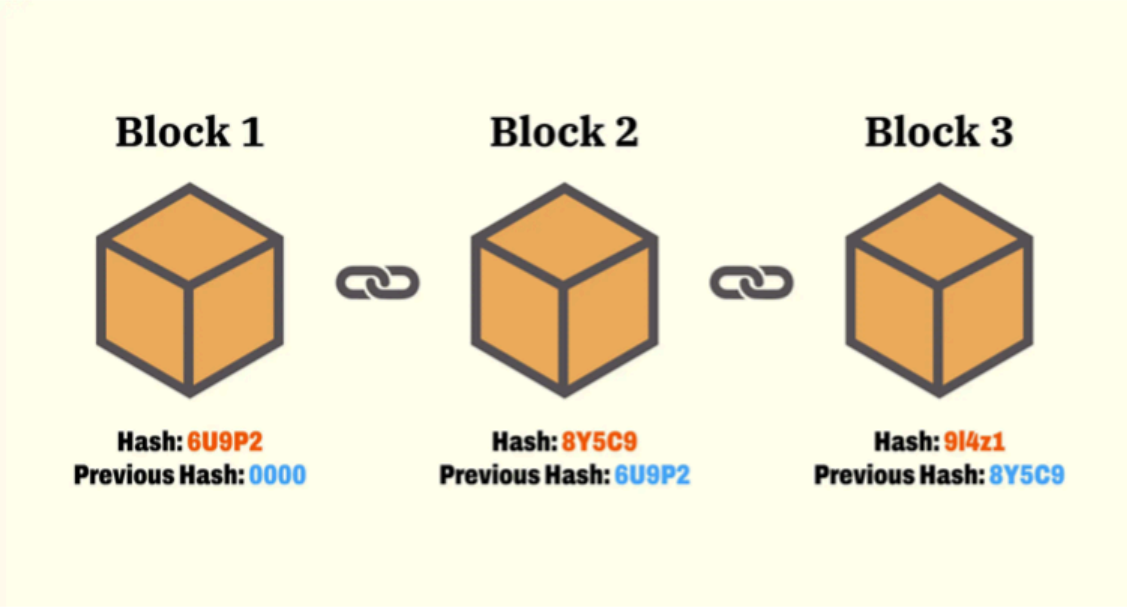
2. **L'hash**: un codice alfanumerico unico, come un'impronta digitale, che identifica quel blocco e i suoi contenuti.

   Se qualcosa cambia nel blocco, l'hash cambia di conseguenza.

3. **L'hash del blocco precedente**: è ciò che collega tra loro i blocchi e forma la catena.

   Questo legame rende l'intero sistema molto più sicuro.

Immaginiamo tre blocchi collegati:



- Il **Blocco 1** è il primo della catena ed è chiamato **Genesis Block**, perché non punta a nessun blocco precedente.
- Il **Blocco 2** contiene il proprio hash e l'hash del Blocco 1.
- Il **Blocco 3** contiene il proprio hash e quello del Blocco 2.

Se un hacker prova a modificare i dati nel Blocco 2, l'hash cambia automaticamente. Di conseguenza, anche il Blocco 3 diventa invalido perché non trova più il riferimento corretto. Questo effetto a catena rende le manomissioni molto difficili.

Gli hash da soli non bastano. I computer moderni sono velocissimi e potrebbero ricalcolare gli hash di tutta la catena in poco tempo.
Per rendere più difficile l'attacco, la blockchain utilizza un sistema chiamato **Proof of Work (PoW)**, cioè "prova di lavoro".

La PoW richiede calcoli matematici complessi e dispendiosi in termini di tempo ed energia.

Nel caso di Bitcoin, servono circa **10 minuti per aggiungere un nuovo blocco**.

Se qualcuno volesse modificare un blocco, dovrebbe rifare la Proof of Work non solo di quel blocco ma anche di tutti quelli successivi: un'impresa praticamente impossibile.

Un altro aspetto chiave è che la blockchain **non è gestita da un'autorità centrale**, ma da una rete di computer collegati tra loro, chiamata **rete peer-to-peer**.

Ogni partecipante della rete (detto **nodo**) possiede una copia completa della blockchain.
Quando un nuovo blocco viene creato, è condiviso con tutti i nodi.
Ogni nodo controlla la validità del blocco e, se è corretto, lo aggiunge alla propria copia.
Se invece il blocco è stato manomesso, viene rifiutato.

Per alterare davvero la blockchain, bisognerebbe controllare contemporaneamente **più del 50% della rete**: un obiettivo estremamente costoso e impraticabile.

Oggi la blockchain viene usata soprattutto per le criptovalute come Bitcoin, ma le sue potenzialità vanno molto oltre:

Smart contracts: contratti digitali che si attivano automaticamente al verificarsi di determinate condizioni.
Sanità: archiviazione sicura delle cartelle cliniche.
Arte digitale: creazione e certificazione di opere uniche tramite i cosiddetti **NFT (Non Fungible Tokens)**.

In generale, qualsiasi informazione che richieda **sicurezza, trasparenza e immutabilità** può essere gestita attraverso una blockchain.

L'output del programma evidenzia che, a seguito della modifica dei dati contenuti nel blocco 1, l'hash associato al blocco non viene aggiornato di conseguenza. Questo comporta una discrepanza nella catena, compromettendo l'integrità e la validità complessiva della blockchain.

ATTIVITÀ PRATICA N. 1

Creare una blockchain semplice

L'obiettivo dell'attività è scrivere il codice per creare una blockchain semplice in modo da capirne la struttura. Ecco il programma in Python *blockchain.py* che realizza una blockchain in memoria.

```
# Importa la libreria hashlib per calcolare hash crittografici
import hashlib

# Classe che rappresenta un singolo blocco della blockchain
class Block:
    def __init__(self, index, data, previous_hash):
        self.index = index  # Posizione del blocco nella catena
        self.data = data  # Dati contenuti nel blocco (es. transazioni)
        self.previous_hash = previous_hash  # Hash del blocco precedente
        self.hash = self.calculate_hash()  # Hash del blocco corrente, calcolato all'istanziazione

    def calculate_hash(self):
        # Crea una stringa unica con i dati del blocco e calcola l'hash SHA-256
        block_string = f"{self.index}{self.data}{self.previous_hash}"
        return hashlib.sha256(block_string.encode()).hexdigest()

# Classe che rappresenta la blockchain (catena di blocchi)
class Blockchain:
    def __init__(self):
        # Inizializza la catena con il blocco genesi
        self.chain = [self.create_genesis_block()]
```

```
    def create_genesis_block(self):
  # Crea il primo blocco della catena, detto "blocco genesi"
        return Block(0, "Blocco iniziale", "0")

    def add_block(self, data):
        # Aggiunge un nuovo blocco alla catena
        previous_block = self.chain[-1]  # Prende l'ultimo blocco della catena
        new_block = Block(len(self.chain), data, previous_block.hash)  # Crea il nuovo blocco
        self.chain.append(new_block)  # Aggiunge il nuovo blocco alla catena

# Demo di utilizzo della blockchain
your_chain = Blockchain()
your_chain.add_block("Alice paga Bob 5€")
your_chain.add_block("Bob paga Carlo 3€")

# Stampa i blocchi della catena con le informazioni principali
for block in your_chain.chain:
    print(block.index, block.data, block.hash[:10], block.previous_hash[:10], "...")
```

L'output del programma è il seguente:

```
0 Blocco iniziale e0c1e59c1c 0 ...
1 Alice paga Bob 5€ ac908a075f e0c1e59c1c ...
2 Bob paga Carlo 3€ e6edc13dbe ac908a075f ...
```

ATTIVITÀ PRATICA N. 2

Verificare la validità di una blockchain

L'obiettivo dell'attività aggiungere il controllo della validità della blockchain calcolando e verificando tutti gli hash dei blocchi prima e dopo la modifica dell'informazione di un blocco. Ecco le modifiche da apportare al codice per ottenere il file *blockchain_check.py*

```
# Classe che rappresenta un singolo blocco della blockchain
class Block:
    def is_valid(self, previous_block):
        """
        Verifica la validità del blocco rispetto al blocco precedente:
        - L'hash calcolato deve corrispondere a quello memorizzato
        - L'hash del blocco precedente deve essere corretto
        """
        return (
            self.hash == self.calculate_hash() and
            self.previous_hash == previous_block.hash
        )
...
# Classe che rappresenta la blockchain (catena di blocchi)
class Blockchain:
    ...

    def is_valid(self):
# Verifica l'integrità della blockchain
        for i in range(1, len(self.chain)):
            current = self.chain[i]
```

```
            previous = self.chain[i-1]
        # Usa il metodo is_valid del blocco
            if not current.is_valid(previous):
                return False
        return True

    # Demo di utilizzo della blockchain
    your_chain = Blockchain()
    your_chain.add_block("Alice paga Bob 5€")
    your_chain.add_block("Bob paga Carlo 3€")

    print("Blockchain valida?", your_chain.is_valid())

    # Stampa i blocchi della catena con le informazioni principali
    for block in your_chain.chain:
        print(block.index, block.data, block.hash[:10], block.previous_hash[:10], "...")


    # Alteriamo i dati di un blocco per simulare un attacco
    your_chain.chain[1].data = "Alice paga Bob 1000€"  # Modifica non autorizzata

    print("Blockchain valida dopo alterazione?", your_chain.is_valid())

    # Stampa i blocchi della catena con le informazioni principali
    for block in your_chain.chain:
        print(block.index, block.data, block.hash[:10], block.previous_hash[:10], "...")
```

L'output del programma evidenzia che, a seguito della modifica dei dati contenuti nel blocco 1, l'hash associato al blocco non viene aggiornato di conseguenza. Questo comporta una discrepanza nella catena, compromettendo l'integrità e la validità complessiva della blockchain.

```
Blockchain valida? True
0 Blocco iniziale e0c1e59c1c 0 ...
1 Alice paga Bob 5€ ac908a075f e0c1e59c1c ...
2 Bob paga Carlo 3€ e6edc13dbe ac908a075f ...
Blockchain valida dopo alterazione? False
0 Blocco iniziale e0c1e59c1c 0 ...
1 Alice paga Bob 1000€ ac908a075f e0c1e59c1c ...
2 Bob paga Carlo 3€ e6edc13dbe ac908a075f ...
```

ATTIVITÀ PRATICA N. 3

Ricostruzione della blockchain

Si propone di aggiungere un metodo `rebuild_chain_from` alla classe `BlockChain` per ricostruire gli hash della catena a partire da un blocco modificato. Tuttavia, questa procedura risulta inefficace in termini di sicurezza se non viene affiancata dall'implementazione di un meccanismo di **proof of work**, poiché i computer moderni sono in grado di ricalcolare rapidamente gli hash di tutti i blocchi della catena. Di seguito viene riportato il codice proposto:

```
def rebuild_chain_from(self, start_index=1):
    """
    Ricostruisce la catena a partire da start_index:
    aggiorna previous_hash e hash di tutti i blocchi successivi.
    """
```

```
        for i in range(start_index, len(self.chain)):
            previous_block = self.chain[i-1]
            self.chain[i].previous_hash = previous_block.hash
            self.chain[i].hash = self.chain[i].calculate_hash()
```

ATTIVITÀ PRATICA N. 4

Implementazione della proof of work

La **Proof of Work** è un meccanismo che richiede di risolvere un problema computazionale difficile per poter validare un blocco. Lo scopo principale è rendere costoso, in termini di tempo e risorse, modificare la blockchain, proteggendola da manomissioni. La funzione `proof_of_work` simula il mining: cerca un **nonce** che produca un hash con un prefisso specifico di zeri. Questo è il cuore della PoW: creare un lavoro computazionale significativo che attesti la validità di un blocco. Il codice *proofofwork.py*:

```
import hashlib
import time

def proof_of_work(message, difficulty):
    """
    Esegue una Proof of Work.

    Args:
        message (str): Il messaggio da "minare"
        difficulty (int): Numero di zeri iniziali richiesti nell'hash

    Returns:
        tuple: (nonce trovato, hash)
    """
    nonce = 0
    prefix = '0' * difficulty
    while True:
        text = f"{message}{nonce}"
        hash_result = hashlib.sha256(text.encode()).hexdigest()
        if hash_result.startswith(prefix):
            return nonce, hash_result
        nonce += 1

# Esempio di utilizzo
message = "Hello, blockchain!"
difficulty = 4  # richiede che l'hash inizi con 4 zeri

start_time = time.time()
nonce, hash_found = proof_of_work(message, difficulty)
end_time = time.time()

print(f"Nonce trovato: {nonce}")
print(f"Hash: {hash_found}")
print(f"Tempo impiegato: {end_time - start_time:.2f} secondi")
```

Proviamo ad eseguire il codice con diversi gradi di difficulty: noteremo come il tempo necessario per risolvere il problema cresca rapidamente. Il numero medio di tentativi per trovare il giusto hash è 16n dove n è il numero di zeri inziali richiesti (ovvero il grado di difficulty), dunque cresce esponenzialmente.

Proviamo a eseguire il codice con diversi livelli di difficoltà. Osserveremo come il tempo necessario per risolvere il problema aumenti rapidamente al crescere della difficoltà. Il numero medio di tentativi richiesti per trovare l'hash corretto è 16n, dove n

rappresenta il numero di zeri iniziali richiesti (cioè il livello di difficoltà). Di conseguenza, il numero di tentativi cresce in modo esponenziale all'aumentare di n.

Difficulty = 5

```
Nonce trovato: 125331
Hash: 0000088c28838cc0fe22118087d9c935aee61a8c1abb2a6ae91c6cc2a78ba2a4
Tempo impiegato: 0.05 secondi
```

Difficulty = 6

```
Nonce trovato: 11692870
Hash: 000000d841d6dbea85c3aca9c91871193486e935afd2ee9b3838b20d9645bfb0
Tempo impiegato: 4.55 secondi
```

Difficulty = 7

```
Nonce trovato: 39460347
Hash: 0000000d0399b15b38cfa54e3816992381ac04f112caeeca7fc8c31b166a4958
Tempo impiegato: 15.36 secondi
```

Difficulty = 8

Nonce trovato: 5827211615
Hash: 000000004861a4eae2b804a48e71e741c1d48218f169634bbf0f69901ba500e5
Tempo impiegato: 7420.97 secondi

ATTIVITÀ PRATICA N. 5

Creazione di una blockchain con PoW

Come attività conclusiva, realizziamo un semplice esempio di **blockchain** che integra il calcolo dell'hash con la **Proof of Work** definita in precedenza. In questo modo possiamo osservare concretamente come, al crescere della *difficulty*, la creazione dei blocchi diventi progressivamente più lenta e onerosa.
Questa caratteristica, come già detto, rende molto costoso, in termini di tempo e risorse computazionali, tentare di modificare retroattivamente la blockchain, contribuendo così alla sua sicurezza e protezione dalle manomissioni.

Codice del file *block_pow.py*:

```
import hashlib
import time

class Block:
    def __init__(self, index, previous_hash, data, difficulty):
# Inizializza le proprietà del blocco
        self.index = index
        self.timestamp = time.time()  # Timestamp di creazione del blocco
        self.data = data
        self.previous_hash = previous_hash
        self.difficulty = difficulty  # Difficoltà del proof-of-work
# Calcola nonce e hash validi tramite proof-of-work
        self.nonce, self.hash = self.proof_of_work()

    def compute_hash(self, nonce):
# Crea una stringa rappresentativa del blocco e calcola il suo hash SHA-256
```

```
        block_string = f"{self.index}{self.timestamp}{self.data}{self.previous_hash}{nonce}"
        return hashlib.sha256(block_string.encode()).hexdigest()

    def proof_of_work(self):
# Esegue il proof-of-work trovando un nonce che produce un hash con un certo numero di zeri iniziali
        nonce = 0
        prefix = '0' * self.difficulty  # Prefisso richiesto per l'hash
        while True:
            hash_result = self.compute_hash(nonce)
            if hash_result.startswith(prefix):
# Restituisce il nonce e l'hash validi
                return nonce, hash_result
            nonce += 1  # Prova il prossimo nonce

class Blockchain:
    def __init__(self, difficulty=6):
# Inizializza la blockchain e crea il blocco Genesis
        self.chain = []  # Lista dei blocchi
        self.difficulty = difficulty  # Difficoltà globale per tutti i blocchi
        self.create_genesis_block()

    def create_genesis_block(self):
# Crea il primo blocco della catena (Genesis Block)
        genesis_block = Block(0, "0", "Genesis Block", self.difficulty)
        self.chain.append(genesis_block)

    def add_block(self, data):
# Aggiunge un nuovo blocco alla catena con i dati forniti
        previous_hash = self.chain[-1].hash  # Hash dell'ultimo blocco
        new_block = Block(len(self.chain), previous_hash, data, self.difficulty)
        self.chain.append(new_block)

    def display_chain(self):
# Stampa tutte le informazioni dei blocchi nella blockchain
        for block in self.chain:
            print(f"Index: {block.index}")
            print(f"Timestamp: {time.ctime(block.timestamp)}")
            print(f"Data: {block.data}")
            print(f"Previous Hash: {block.previous_hash}")
            print(f"Hash: {block.hash}")
            print(f"Nonce: {block.nonce}")
            print("-" * 30)

# Esempio di utilizzo
blockchain = Blockchain(difficulty=6)
blockchain.add_block("Primo blocco dopo Genesis")
blockchain.add_block("Secondo blocco dopo Genesis")
blockchain.display_chain()
```

L'output del programma mostra il processo di costruzione della blockchain composta da soli due blocchi. Durante l'esecuzione è possibile osservare il tempo effettivamente impiegato per generare la catena. Per rendere l'esperimento ancora più interessante, si può provare a modificare un'informazione all'interno della blockchain e tentare di ricostruirla: in questo modo si sperimenta direttamente quanto tempo e quante risorse siano necessarie per ristabilire la validità della catena (*blockchain_changeall.py*).

Index: 0
Timestamp: Sat Sep 6 17:34:35 2025
Data: Genesis Block
Previous Hash: 0
Hash: 000000b1a48c74d918535da0f3a41e4afa4351ea723a58fa2d99a99fa41986f4
Nonce: 4531869

Index: 1
Timestamp: Sat Sep 6 17:34:38 2025
Data: Primo blocco dopo Genesis
Previous Hash: 000000b1a48c74d918535da0f3a41e4afa4351ea723a58fa2d99a99fa41986f4
Hash: 000000e5a66e6e9e65b2d6ef96c8f64fc75bd68dfe01746668dd0d38c2ff31c5
Nonce: 19932150

Index: 2
Timestamp: Sat Sep 6 17:34:52 2025
Data: Secondo blocco dopo Genesis
Previous Hash: 000000e5a66e6e9e65b2d6ef96c8f64fc75bd68dfe01746668dd0d38c2ff31c5
Hash: 000000eec4ee7785ede5e186113a0c51348def17cd8ecfb7c83e7deefedab5b4
Nonce: 5097957
