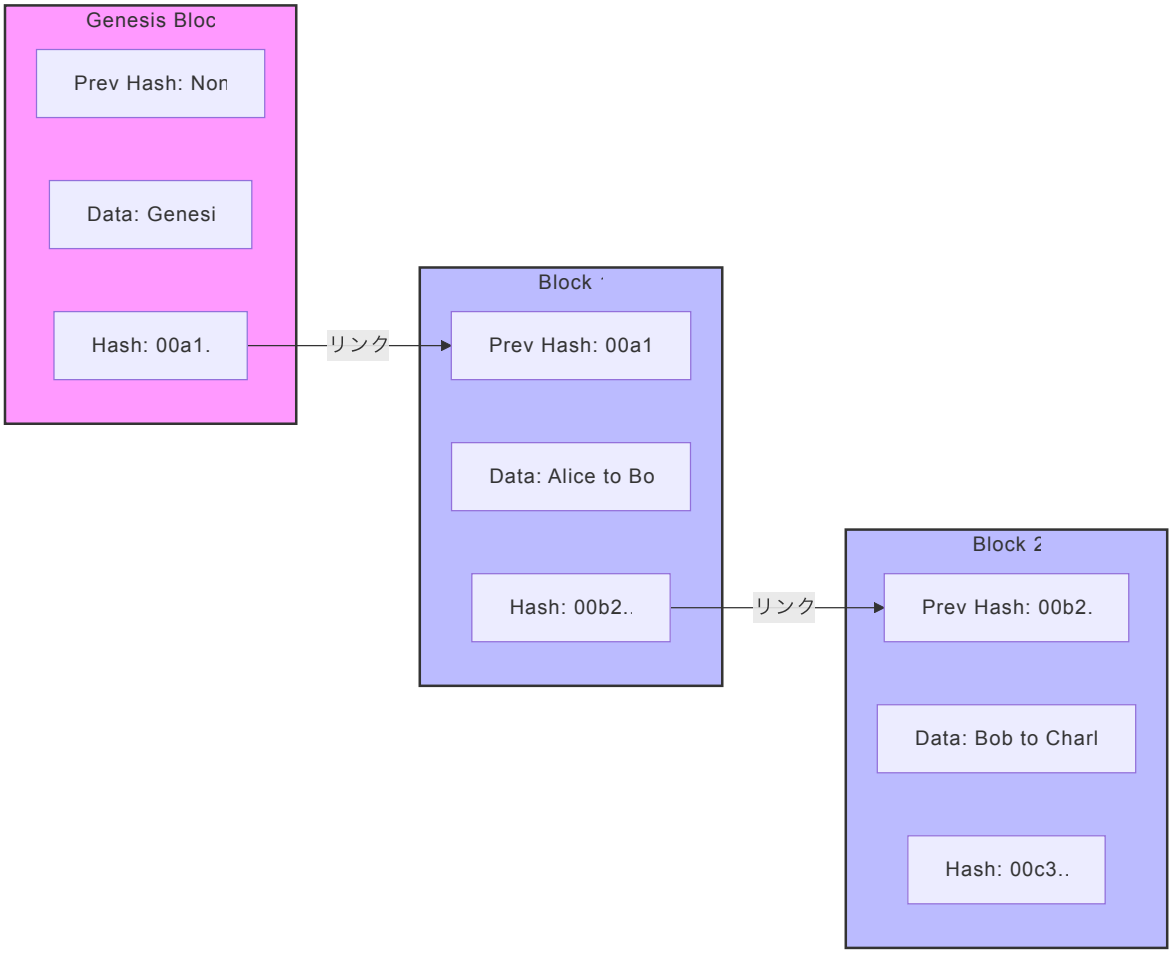
Pythonで作る簡易ブロックチェーン

ブロックチェーンの本質は **「改ざんが極めて困難な分散型台帳」** です。これを実現するための重要な3つの概念をコードに落とし込みます。

1. **Block (ブロック)** : データと前のブロックへのリンクを持つ箱。
2. **Chain (チェーン)** : ブロックをハッシュ値で鎖のように繋ぐ構造。
3. **Proof of Work (PoW)** : 不正を防ぐための計算コスト（採掘/マイニング）。

1. 構造の可視化 (Mermaid)

各ブロックは **「前のブロックのハッシュ値」** を含んでいます。これにより、過去のデータを少しでも書き換えると、それ以降のすべてのハッシュ値が整合しなくなるため、改ざんが検出できます。



---

2. Pythonによる実装

外部ライブラリを使わず標準ライブラリ（ `hashlib` , `json` , `time` ）のみで実装します。

Step 1: Blockクラスの作成

ブロック単体を定義します。ここには **「ナンス (Nonce)」** という値を含めます。これは後述するマイニング（PoW）で使用する「使い捨ての数値」です。

```
import hashlib
import json
import time

class Block:
  def __init__(self, index, transactions, timestamp, previous_hash):
```

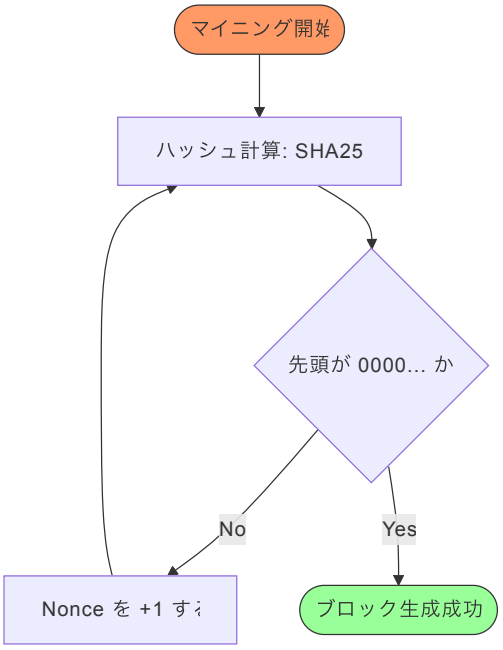
```
    self.index = index
    self.transactions = transactions  # 取引データ（例: 送金記録など）
    self.timestamp = timestamp
    self.previous_hash = previous_hash
    self.nonce = 0                    # マイニングで調整する数値
    self.hash = self.compute_hash()   # 自身のハッシュ値を計算

  def compute_hash(self):
    """
    ブロックの内容（メタデータ含む）をJSON文字列にし、SHA-256ハッシュを計算する
    """
    block_string = json.dumps(self.__dict__, sort_keys=True)
    return hashlib.sha256(block_string.encode()).hexdigest()
```

Step 2: Blockchainクラスとマイニング (PoW)

次に、ブロックを管理するチェーン本体を作成します。ここで重要なのが **プルーフ・オブ・ワーク (PoW)** です。

PoWの仕組み:
ハッシュ値の先頭に特定の数（難易度）だけ「0」が並ぶようなハッシュ値を見つけるまで、 `nonce` を変えながら計算を繰り返します。

マイニングのフローチャート:



コード実装:

```
class Blockchain:
  def __init__(self):
    self.unconfirmed_transactions = [] # まだブロックに入っていない取引
    self.chain = []
    self.difficulty = 4 # 先頭に0が4つ並ぶハッシュを見つける（難易度）
    self.create_genesis_block()

  def create_genesis_block(self):
    """
    最初のブロック（ジェネシスブロック）を生成してチェーンに追加
    """
    genesis_block = Block(0, [], time.time(), "0")
    genesis_block.hash = genesis_block.compute_hash()
    self.chain.append(genesis_block)

  @property
  def last_block(self):
    return self.chain[-1]

  def proof_of_work(self, block):
    """
    マイニング処理：
    ハッシュ値の先頭が difficulty の数だけ '0' になるまで nonce を増やして再計算する
    """
    computed_hash = block.compute_hash()
    while not computed_hash.startswith('0' * self.difficulty):
```

```
            block.nonce += 1
            computed_hash = block.compute_hash()
        return computed_hash

    def add_new_transaction(self, transaction):
        """
        新しい取引データをプールに追加
        """
        self.unconfirmed_transactions.append(transaction)

    def mine(self):
        """
        未承認取引をまとめてブロックを作成し、マイニングを行ってチェーンに追加する
        """
        if not self.unconfirmed_transactions:
            return False

        last_block = self.last_block

        new_block = Block(index=last_block.index + 1,
                          transactions=self.unconfirmed_transactions,
                          timestamp=time.time(),
                          previous_hash=last_block.hash)

        # ここで計算処理（PoW）が走る
        print(f"Mining block {new_block.index}...")
        proof = self.proof_of_work(new_block)

        # マイニング成功後、ハッシュを確定させてチェーンに追加
        new_block.hash = proof
        self.chain.append(new_block)

        # トランザクションプールを空にする
        self.unconfirmed_transactions = []
        return new_block.index
```

Step 3: 実行してみる

実際にブロックチェーンを動かして、マイニングとチェーンの接続を確認します。

```
# 1. ブロックチェーンのインスタンス化
my_blockchain = Blockchain()

print("--- Start Blockchain Demo ---")

# 2. トランザクション（取引）の追加
my_blockchain.add_new_transaction({"sender": "UserA", "receiver": "UserB", "amount": 10})
my_blockchain.add_new_transaction({"sender": "UserB", "receiver": "UserC", "amount": 5})

# 3. マイニング実行（ブロック1）
my_blockchain.mine()

# 4. さらに取引を追加
my_blockchain.add_new_transaction({"sender": "UserC", "receiver": "UserA", "amount": 2})

# 5. マイニング実行（ブロック2）
my_blockchain.mine()

print("\n--- Blockchain Result ---")
for block in my_blockchain.chain:
    print(f"Index: {block.index}")
    print(f"Transactions: {block.transactions}")
    print(f"Timestamp: {block.timestamp}")
    print(f"Nonce: {block.nonce}")     # 正解を見つけるために試行錯誤した回数
    print(f"Prev Hash: {block.previous_hash}")
    print(f"Hash:      {block.hash}") # 必ず '0000' で始まっているはず
    print("-" * 30)
```

3. 実行結果の解説

上記のコードを実行すると、以下のような出力が得られます（ハッシュ値等は実行ごとに異なります）。

- **Hashの先頭**：`difficulty = 4` に設定したため、 `Hash` は必ず `0000` で始まっています。

- **Nonce** : `0000` で始まるハッシュを見つけるために、計算された数値が入っています（例: 45212など）。
- **Prev Hash** : Index 2の `Prev Hash` は、Index 1の `Hash` と完全に一致しており、鎖（チェーン）が繋がっていることがわかります。