#Task 2: Write a program in Python for mining a new block in a blockchain, and print the values of the new block.

```
import hashlib
import json
import time
from typing import List

class Block:
    def __init__(self, index, previous_hash, timestamp, data, hash, nonce=0):
        self.index = index
        self.previous_hash = previous_hash
        self.timestamp = timestamp
        self.data = data
        self.hash = hash
        self.nonce = nonce

    def __repr__(self):
        return f"Block(index={self.index}, hash={self.hash}, prev_hash={self.previous_hash},
nonce={self.nonce})"


def calculate_hash(index, previous_hash, timestamp, data, nonce):
    return
hashlib.sha256(f"{index}{previous_hash}{timestamp}{data}{nonce}".encode()).hexdigest()


def create_genesis_block():
    return Block(0, "0", int(time.time()), "Genesis Block", calculate_hash(0, "0", int(time.time()),
"Genesis Block", 0))


def mine_block(previous_block, data, difficulty=4):
    index = previous_block.index + 1
    timestamp = int(time.time())
    nonce = 0
    hash_value = calculate_hash(index, previous_block.hash, timestamp, data, nonce)
    while not hash_value.startswith("0" * difficulty):
        nonce += 1
        hash_value = calculate_hash(index, previous_block.hash, timestamp, data, nonce)
    return Block(index, previous_block.hash, timestamp, data, hash_value, nonce)


# Task 1: Implement a Blockchain
```

```
class Blockchain:
    def __init__(self):
        self.chain: List[Block] = [create_genesis_block()]

    def add_block(self, data):
        new_block = mine_block(self.chain[-1], data)
        self.chain.append(new_block)

    def print_chain(self):
        for block in self.chain:
            print(block)
            print("-" * 50)


# Creating blockchain and mining blocks
blockchain = Blockchain()
print()
blockchain.add_block("First Block")
blockchain.add_block("Second Block")
blockchain.print_chain()
```

#Task 4: Write a program in Python to implement a blockchain and print the values of all fields as described in etherscan.io import hashlib

```
import hashlib
import datetime

class Block:
    def __init__(self, block_number, transactions, previous_hash, gas_limit, gas_used, miner):
        self.block_number = block_number
        self.timestamp = datetime.datetime.now()
        self.transactions = transactions
        self.previous_hash = previous_hash
        self.gas_limit = gas_limit
        self.gas_used = gas_used
        self.miner = miner
        self.hash = self.calculate_hash()

    def calculate_hash(self):
        """
        Calculate SHA-256 hash of the block based on its attributes.
        """
```

```
        data_string = (
            str(self.block_number) +
            str(self.timestamp) +
            str(self.transactions) +
            str(self.previous_hash) +
            str(self.gas_limit) +
            str(self.gas_used) +
            str(self.miner)
        )
        return hashlib.sha256(data_string.encode('utf-8')).hexdigest()

class Blockchain:
    def __init__(self):
        """
        Initialize the blockchain with a Genesis Block.
        """
        self.chain = [self.create_genesis_block()]

    def create_genesis_block(self):
        """
        Create the first block in the blockchain (Genesis Block).
        """
        return Block(0, "Genesis Block", "0", 0, 0, "Genesis Miner")

    def add_block(self, new_block):
        """
        Add a new block to the blockchain.
        """
        new_block.previous_hash = self.chain[-1].hash  # Link to previous block
        new_block.hash = new_block.calculate_hash()  # Calculate the new block's hash
        self.chain.append(new_block)

    def print_block(self, block):
        """
        Print all details of a given block.
        """
        print(f"Block Number: {block.block_number}")
        print(f"Timestamp: {block.timestamp}")
        print(f"Transactions: {block.transactions}")
        print(f"Previous Hash: {block.previous_hash}")
        print(f"Gas Limit: {block.gas_limit}")
        print(f"Gas Used: {block.gas_used}")
        print(f"Miner: {block.miner}")
        print(f"Hash: {block.hash}")
```

```
        print("-" * 50)

    def traverse_chain(self):
        """
        Print all blocks in the blockchain.
        """
        for block in self.chain:
            self.print_block(block)


# === Creating the Blockchain ===
my_blockchain = Blockchain()

# Adding new blocks with transactions, gas limits, and miner information
my_blockchain.add_block(Block(1, "Transaction 1", "", 1000000, 500000, "Miner 1"))
my_blockchain.add_block(Block(2, "Transaction 2", "", 2000000, 1500000, "Miner 2"))
my_blockchain.add_block(Block(3, "Transaction 3", "", 3000000, 2500000, "Miner 3"))

# Print all blocks
my_blockchain.traverse_chain()
```

 # Task 6: Implementing Proof of Work (PoW) Algorithm

```
import hashlib
import time

class Block:
    def __init__(self, data, previous_hash):
        """
        Initializes a new block.
        """
        self.timestamp = time.time()
        self.data = data
        self.previous_hash = previous_hash
        self.nonce = 0  # Starts at 0 and increases during mining
        self.hash = self.generate_hash()

    def generate_hash(self):
        """
        Generates a SHA-256 hash of the block's contents.
        """
        block_contents = str(self.timestamp) + str(self.data) + str(self.previous_hash) +
str(self.nonce)
```

```
        block_hash = hashlib.sha256(block_contents.encode()).hexdigest()
        return block_hash

    def mine_block(self, difficulty):
        """
        Implements Proof of Work (PoW): Finds a valid hash with required leading zeros.
        """
        while self.hash[:difficulty] != "0" * difficulty:
            self.nonce += 1
            self.hash = self.generate_hash()

        print(f"Block mined: {self.hash}")

class Blockchain:
    def __init__(self):
        """
        Initializes the blockchain with a Genesis Block.
        """
        self.chain = [self.create_genesis_block()]
        self.difficulty = 2  # Set PoW difficulty level

    def create_genesis_block(self):
        """
        Creates the first block (Genesis Block).
        """
        return Block("Genesis Block", "0")

    def get_latest_block(self):
        """
        Returns the most recently added block in the blockchain.
        """
        return self.chain[-1]

    def add_block(self, new_block):
        """
        Mines and adds a new block to the blockchain.
        """
        new_block.previous_hash = self.get_latest_block().hash  # Link new block to previous block
        new_block.mine_block(self.difficulty)  # Perform mining
        self.chain.append(new_block)

    def is_chain_valid(self):
        """
```

```
        Validates the blockchain by checking hashes and previous hash links.
        """
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i - 1]

            # Check if hash is correct
            if current_block.hash != current_block.generate_hash():
                return False

            # Check if previous_hash matches actual previous block's hash
            if current_block.previous_hash != previous_block.hash:
                return False

        return True

# === Running the Blockchain with Proof-of-Work ===
if __name__ == "__main__":
    blockchain = Blockchain()

    print("\nMining block 1...")
    block1 = Block("Transaction 1", "")
    blockchain.add_block(block1)

    print("\nMining block 2...")
    block2 = Block("Transaction 2", "")
    blockchain.add_block(block2)

    print("\nMining block 3...")
    block3 = Block("Transaction 3", "")
    blockchain.add_block(block3)

    # Validate blockchain integrity
    print("\nIs blockchain valid? {}".format(blockchain.is_chain_valid()))

    # Tampering with blockchain data
    blockchain.chain[1].data = "Tampered transaction"

    # Revalidate blockchain after tampering
    print("\nIs blockchain valid after tampering? {}".format(blockchain.is_chain_valid()))
```

Task 8: Write a program in Python to Fetch the Latest Block Information from Ethereum Blockchain Using Etherscan API

```
import requests

def get_latest_block(api_key):
    """
    Fetch the latest block information from Ethereum blockchain using Etherscan API.
    """
    url = "https://api.etherscan.io/api"

    # Define API request parameters
    params = {
        "module": "proxy",  # Access the Ethereum JSON-RPC API via Etherscan
        "action": "eth_getBlockByNumber",  # Fetch block details by number
        "tag": "latest",  # Get the latest block
        "boolean": "true",  # Return full transaction details
        "apikey": api_key,  # Your Etherscan API key
    }

    try:
        # Send GET request to Etherscan API
        response = requests.get(url, params=params)

        # Check if the request was successful (status code 200)
        if response.status_code == 200:
            data = response.json()  # Convert response to JSON format
            return data["result"]  # Extract block information
        else:
            print("Request failed with status code:", response.status_code)

    except requests.RequestException as e:
        print("Request failed:", str(e))

    return None  # Return None if the request fails

# Replace "YOUR_API_KEY" with your actual Etherscan API key
api_key = "E34342B4IR3B8RI3K61XG4YKEUT7SR54MM"

# Fetch the latest block details
latest_block = get_latest_block(api_key)

# Print block details if successfully fetched
```

```
if latest_block is not None:
    print("\n=== Latest Block Information ===")
    print("Block Number:", int(latest_block["number"], 16))  # Convert hex to decimal
    print("Timestamp:", int(latest_block["timestamp"], 16))
    print("Miner Address:", latest_block["miner"])
    print("Difficulty:", int(latest_block["difficulty"], 16))
    print("Total Difficulty:", int(latest_block["totalDifficulty"], 16))
    print("Gas Limit:", int(latest_block["gasLimit"], 16))
    print("Gas Used:", int(latest_block["gasUsed"], 16))
    print("Transaction Count:", len(latest_block["transactions"]))

    print("\n=== Transactions in Latest Block ===")
    for tx in latest_block["transactions"][:5]:  # Display only first 5 transactions
        print(f"Transaction Hash: {tx['hash']}")
        print(f"From: {tx['from']}")
        print(f"To: {tx['to']}")
        print(f"Value (in Wei): {tx['value']}")
        print("-" * 50)
else:
    print("Failed to fetch the latest block information.")
```

#Task 10: Write a program in Python that Demonstrates How to Use the SHA-256 Hash Function and Its Application in a Simple Blockchain

```
import hashlib
import json
from time import time

class Block:
    def __init__(self, index, timestamp, data, previous_hash):
        """
        Initializes a block with index, timestamp, data, and previous hash.
        """
        self.index = index
        self.timestamp = timestamp
        self.data = data
        self.previous_hash = previous_hash
        self.hash = self.calculate_hash()  # Compute the hash at creation

    def calculate_hash(self):
        """
        Calculates SHA-256 hash of the block using its attributes.
        """
```

```
        block_string = json.dumps({
            "index": self.index,
            "timestamp": self.timestamp,
            "data": self.data,
            "previous_hash": self.previous_hash
        }, sort_keys=True)

        return hashlib.sha256(block_string.encode()).hexdigest()

class Blockchain:
    def __init__(self):
        """
        Initializes the blockchain with a Genesis Block.
        """
        self.chain = [self.create_genesis_block()]

    def create_genesis_block(self):
        """
        Creates the first block of the blockchain (Genesis Block).
        """
        return Block(0, time(), "Genesis Block", "0")

    def add_block(self, data):
        """
        Mines and adds a new block to the blockchain.
        """
        previous_block = self.chain[-1]
        new_block = Block(previous_block.index + 1, time(), data, previous_block.hash)

        # Recalculate hash after setting previous_hash
        new_block.hash = new_block.calculate_hash()

        self.chain.append(new_block)

    def is_chain_valid(self):
        """
        Validates the blockchain by checking hashes and previous hash links.
        """
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i - 1]

            # Recalculate hash and compare with stored hash
            if current_block.hash != current_block.calculate_hash():
```

```
            print(f"Block {current_block.index} has an invalid hash.")
            return False

        # Check if previous_hash value matches actual previous block's hash
        if current_block.previous_hash != previous_block.hash:
            print(f"Block {current_block.index} has an invalid previous hash.")
            return False

    return True

# === Running the Blockchain ===
blockchain = Blockchain()

# Adding blocks with transaction data
blockchain.add_block("Transaction 1")
blockchain.add_block("Transaction 2")
blockchain.add_block("Transaction 3")

# Check if the blockchain is valid
print("Blockchain is valid:", blockchain.is_chain_valid())

# Tampering with the second block
blockchain.chain[1].data = "Tampered Transaction"

# Revalidate the blockchain after tampering
print("Blockchain is valid after tampering:", blockchain.is_chain_valid())
```

 # Task 12: Write a Python program to Demonstrate a Simple Implementation of a Blockchain Using Hash Codes as a Chain of Blocks

```
import hashlib
import datetime

class Block:
    def __init__(self, timestamp, data, previous_hash):
        """
        Initializes a new block with a timestamp, data, and previous block hash.
        """
        self.timestamp = timestamp
        self.data = data
        self.previous_hash = previous_hash
        self.hash = self.calculate_hash()
```

```
    def calculate_hash(self):
        """
        Generates a SHA-256 hash of the block's contents.
        """
        hash_string = str(self.timestamp) + str(self.data) + str(self.previous_hash)
        return hashlib.sha256(hash_string.encode()).hexdigest()

class Blockchain:
    def __init__(self):
        """
        Initializes the blockchain with a Genesis Block.
        """
        self.chain = [self.create_genesis_block()]

    def create_genesis_block(self):
        """
        Creates the first block of the blockchain (Genesis Block).
        """
        return Block(datetime.datetime.now(), "Genesis Block", "0")

    def get_latest_block(self):
        """
        Returns the most recently added block in the blockchain.
        """
        return self.chain[-1]

    def add_block(self, new_block):
        """
        Adds a new block to the blockchain.
        """
        new_block.previous_hash = self.get_latest_block().hash  # Link the new block
        new_block.hash = new_block.calculate_hash()  # Recalculate hash
        self.chain.append(new_block)

    def is_valid(self):
        """
        Validates the blockchain by checking hashes and previous hash links.
        """
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i - 1]

            # Check if the block's hash is correct
            if current_block.hash != current_block.calculate_hash():
```

```
            return False

        # Check if the previous_hash matches the actual previous block's hash
        if current_block.previous_hash != previous_block.hash:
            return False

    return True

# === Creating the Blockchain ===
blockchain = Blockchain()

# Adding new blocks with data
blockchain.add_block(Block(datetime.datetime.now(), "Block 1", ""))
blockchain.add_block(Block(datetime.datetime.now(), "Block 2", ""))
blockchain.add_block(Block(datetime.datetime.now(), "Block 3", ""))

# Check if the blockchain is valid
print("Is blockchain valid?", blockchain.is_valid())

# === Tampering with the blockchain ===
blockchain.chain[1].data = "Modified Block"

# Check blockchain validity after tampering
print("Is manipulated blockchain valid?", blockchain.is_valid())
```

 # Task 14: Write a program in Python to Create a Merkle Tree in Blockchain

```
import hashlib

def hash_node(data):
    """
    Returns the SHA-256 hash of the given data.
    """
    return hashlib.sha256(data.encode()).hexdigest()

def build_merkle_tree(leaves):
    """
    Builds a Merkle Tree from a list of leaves and returns the root.
    Also prints the tree level by level.
    """
    # Hash all leaves first
    tree = [[hash_node(leaf) for leaf in leaves]]
```

```
    # Build the tree level by level
    while len(tree[-1]) > 1:
        current_level = tree[-1]

        # Ensure even number of nodes by duplicating the last node if needed
        if len(current_level) % 2 == 1:
            current_level.append(current_level[-1])

        # Compute the next level by hashing pairs of nodes
        next_level = [hash_node(current_level[i] + current_level[i + 1]) for i in range(0,
len(current_level), 2)]

        tree.append(next_level)

    return tree

def print_merkle_tree(tree):
    """
    Prints the Merkle Tree level by level.
    """
    print("\nMerkle Tree Structure (Level-wise):")
    for level in range(len(tree)):
        print(f"Level {level}: {tree[level]}")

# Example usage
leaves = ["apple", "banana", "cherry", "date"]

# Build the Merkle Tree
merkle_tree = build_merkle_tree(leaves)

# Print the tree level by level
print_merkle_tree(merkle_tree)

# Print the Merkle Root (Top Level)
print("\nMerkle Root:", merkle_tree[-1][0])
```