Proof of Work Example

Overview

This notebook develops a minimal Proof of Work (PoW) blockchain in Python.

You will implement the core mechanics step by step:

- define SHA-256 hashing helpers,
- mine a block by searching over nonce values,
- verify block validity and detect tampering,
- link multiple blocks into a chain and verify chain integrity.

The objective is to make PoW mechanics explicit and testable, not abstract.

Core building blocks

A hash function maps any input (text, numbers, files) to a fixed-size output called a digest. Cryptographic hashes are deterministic (same input, same output), highly sensitive to input changes, and hard to reverse.

In PoW systems like Bitcoin, the hash function is SHA-256. It always outputs 256 bits, written as 64 hexadecimal characters. For example, `sha256_hex("hello")` equals `2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824`.

SHA-256 repeatedly mixes bits through many rounds and outputs a value that looks random. That is why PoW mining is brute-force search over nonce values.

Why 64 hexadecimal characters? Each hex character represents 4 bits, so

$$64 \times 4 = 256 \text{ bits.}$$

The helper function below computes that SHA-256 digest for any input string.

```python
import hashlib
import time
import pandas as pd

def sha256_hex(s: str) -> str:
    return hashlib.sha256(s.encode("utf-8")).hexdigest()
```

A block has two parts only:

- `header`: metadata used for PoW (`prev_hash`, `merkle_root`, `timestamp`, `nonce`, `difficulty`)[1]
- `body`: transaction payload (a single transaction string in this toy model)[2]

PoW hashes only the header. The helper below takes a header object and returns its SHA-256 digest.

During mining, `prev_hash`, `merkle_root`, and `timestamp` are fixed for one attempt, and miners vary only the nonce to produce new candidate hashes. Conceptually,

$$h(\text{nonce}) = \text{SHA-256}(\text{prev\_hash} \,\|\, \text{merkle\_root} \,\|\, \text{timestamp} \,\|\, \text{nonce} \,\|\, \text{difficulty}),$$

where `nonce` is an integer and `h()` returns a 64-character hexadecimal string. PoW is a search for a nonce such that $h(\text{nonce})$ satisfies the difficulty rule.

```python
def header_hash(header: dict) -> str:
    serialized = (
        f"{header['prev_hash']}|"
        f"{header['merkle_root']}|"
        f"{header['timestamp']}|"
        f"{header['nonce']}|"
        f"{header['difficulty']}"
    )
    return sha256_hex(serialized)
```

[1] In Bitcoin, the header stores `nBits`, a compact encoding of the target threshold. This notebook uses an integer `difficulty` (leading zero hex characters) as a pedagogical simplification.

[2] In this notebook, the block body is exactly one transaction string, so `merkle_root` is just `sha256(transaction)`. In real blockchains, the body contains many transactions and `merkle_root` is computed from a full Merkle tree.

This function checks difficulty. It returns True only if the hash starts with the required number of leading zero hex characters.

```
def meets_difficulty(hash_hex: str, difficulty: int) -> bool:
    return hash_hex.startswith("0" * difficulty)
```

The mining function now builds header and body explicitly. It brute-forces header["nonce"] until the header hash meets difficulty.

```
def mine_block(
    prev_hash: str,
    tx_summary: str,
    difficulty: int,
    max_tries: int = 5_000_000,
) -> dict:
    body = tx_summary
    merkle_root = sha256_hex(body)
    timestamp = int(time.time())
    for nonce in range(max_tries):
        header = {
            "prev_hash": prev_hash,
            "merkle_root": merkle_root,
            "timestamp": timestamp,
            "nonce": nonce,
            "difficulty": difficulty,
        }
        hash_hex = header_hash(header)
        if meets_difficulty(hash_hex, difficulty):
            return {
                "header": header,
                "body": body,
            }
    raise RuntimeError("Increase max_tries or lower difficulty.")
```

Mine and verify a block

This block initializes a genesis-like previous hash, sets transaction content for the body, mines the first block, and prints the mined block contents.[3]

```
prev_hash = "0" * 64
tx_summary = "Alice pays Bob 1 coin"
required_difficulty = 4
block1 = mine_block(prev_hash, tx_summary, difficulty=required_difficulty)
block1
```

```
{'header': {'prev_hash': '0000000000000000000000000000000000000000000000000000000000000000',
  'merkle_root': '0a77bba4e4457e5f6301818876eeb11b1b19830a0fc6d014b63107c75337e974',
  'timestamp': 1771626489,
  'nonce': 63008,
  'difficulty': 4},
 'body': 'Alice pays Bob 1 coin'}
```

The next function verifies one block by recomputing the header hash, checking difficulty, and checking that the body matches the header's `merkle_root`. In real networks, difficulty comes from consensus rules, not from untrusted block fields, so we pass it in explicitly.

```
def verify_block(block: dict, required_difficulty: int) -> bool:
    header = block["header"]
    body = block["body"]
    recomputed_hash = header_hash(header)
    body_matches_root = sha256_hex(body) == header["merkle_root"]
    difficulty_matches = header["difficulty"] == required_difficulty
    return (
        difficulty_matches
        and meets_difficulty(recomputed_hash, required_difficulty)
        and body_matches_root
```

[3]After a miner finds a valid block, it broadcasts the block to peers. Other nodes independently verify PoW, transaction validity, and consensus rules; if valid, they append it to their best chain and relay it further. A mempool is a node's local queue of unconfirmed transactions, so transactions included in the new block are removed from mempools. Users usually wait for additional blocks (confirmations) before treating a payment as final.

```
    )

verify_block(block1, required_difficulty)
```

```
True
```

This block prints the mined hash and whether verification succeeds.

```
print("Hash:", header_hash(block1["header"]))
print("Valid:", verify_block(block1, required_difficulty))
```

```
Hash: 00004f07073b8b065c723075211d40021f2af040991e0b3edb3e9af288e505b1
Valid: True
```

The final block in this section simulates tampering by changing the body (`tx_summary`) without re-mining, then checks validity to show it fails.

```
tampered = {
    "header": dict(block1["header"]),
    "body": block1["body"],
}
tampered["body"] = "Alice pays Bob 100 coins"

print("Tampered valid?", verify_block(tampered, required_difficulty))
```

```
Tampered valid? False
```

Build and verify a chain

This code extends the single-block example into a true chain structure. Each new block stores the previous block's hash in `header.prev_hash`, so blocks are connected by cryptographic pointers rather than by position in a list.

That linkage is the key security idea: if someone changes an older block, its hash changes, which breaks the `header.prev_hash` reference in the next block. In practice, an attacker would have to re-mine that block and every block after it to restore consistency.

The helper function `link_block` takes the previous block and a new transaction summary string, then mines the next block.

```python
def link_block(prev_block: dict, tx_summary: str, difficulty: int) -> dict:
    prev_hash = header_hash(prev_block["header"])
    return mine_block(prev_hash, tx_summary, difficulty=difficulty)

block0 = {
    "header": {
        "prev_hash": "0" * 64,
        "merkle_root": sha256_hex("genesis"),
        "timestamp": 0,
        "nonce": 0,
        "difficulty": 0,
    },
    "body": "genesis",
}

tx_stream = [
    "Alice pays Bob 1.0 coin",
    "Bob pays Carol 0.2 coins",
    "Carol pays Dave 0.1 coins",
    "Dave pays Erin 0.05 coins",
    "Erin pays Frank 0.03 coins",
    "Frank pays Grace 0.02 coins",
    "Grace pays Heidi 0.01 coins",
    "Heidi pays Ivan 0.01 coins",
]

chain = []
prev = block0
for tx in tx_stream:
    b = link_block(prev, tx, difficulty=required_difficulty)
    chain.append(b)
    prev = b

{"chain_length": len(chain)}
```

```
{'chain_length': 8}
```

The chain verifier enforces three checks in order. First, each block must pass `verify_block`. Second, the first block must link to the expected parent hash (the prior block, here `block0`). Third, every later block must point to the previous block's hash.

This is essentially what full nodes do when they receive blocks: verify PoW and data integrity block-by-block, then verify linkage across the chain. If either check fails at any point, the chain is rejected.

```
def verify_chain(chain, required_difficulty: int, first_prev_hash: str):
    if not chain:
        return True
    for idx, block in enumerate(chain):
        if not verify_block(block, required_difficulty):
            return False
        if idx == 0 and block["header"]["prev_hash"] != first_prev_hash:
            return False
        if idx > 0 and block["header"]["prev_hash"] != header_hash(chain[idx - 1]["header"])
            return False
    return True

verify_chain(chain, required_difficulty, first_prev_hash=header_hash(block0["header"]))
```

```
True
```

Visualize the chain

This final block prints a compact view of the chain so you can see each block's hash, its `header.prev_hash` pointer, and whether each pointer correctly links to the previous block.

```
rows = []
for i, b in enumerate(chain, start=1):
    linked_ok = True if i == 1 else (b["header"]["prev_hash"] == header_hash(chain[i-2]["hea
    rows.append({
        "block": i,
        "nonce": b["header"]["nonce"],
        "hash_prefix": header_hash(b["header"])[:12],
        "prev_hash_prefix": b["header"]["prev_hash"][:12],
        "linked_ok": linked_ok,
    })

pd.DataFrame(rows)
```

	block	nonce	hash_prefix	prev_hash_prefix	linked_ok
0	1	113583	0000d18a5aa4	52cae89ca662	True
1	2	63499	0000e2145915	0000d18a5aa4	True
2	3	4950	000023922552	0000e2145915	True
3	4	156737	0000808d7306	000023922552	True
4	5	77988	0000195d5325	0000808d7306	True
5	6	102734	0000802e011b	0000195d5325	True
6	7	23072	00001bc38ae1	0000802e011b	True
7	8	125285	00005444fc56	00001bc38ae1	True

In this notebook, nonce starts at `0` and increases by `1` on each mining attempt. That means nonce is an attempt index, while tries is an attempt count. Attempt 1 uses nonce `0`, attempt 2 uses nonce `1`, and in general success at nonce `k` means you have made `k + 1` total tries.

So the relationship if nonce starts at 0 is:

$$\text{tries} = \text{nonce} + 1.$$