EN.601.441/641: Assignment 1b

February 2026

This is an pair assignment. Submit to Gradescope by 11:59pm February 25, 2026.

1. Specification. For this program you will write a simulation of a "Minimum Viable Blockchain" (MVB), a simplified version of the technology underlying Bitcoin. Your implementation will include nodes that validate transactions, perform proofs of work, and communicate in order to process transactions. This will emulate the mining and transaction verification process of Bitcoin.

Recommended Reading: We highly encourage you to use *Learn Me a Bitcoin* (`https://learnmeabitcoin.com/`) as a reference throughout this assignment. It provides clear explanations and visualizations for transactions, scripts, Merkle trees, and other concepts you will implement.

Your MVB will implement the following:

- Authentic transactions that are resistant to theft through **Bitcoin Script (P2PKH)** execution
- Open competition amongst nodes to validate transactions
- Detection of double spending via the **UTXO(Unspent Transaction Output)** model
- Use of proof-of-work to raise the cost of running attacks against the network through **Nakamoto Consensus**
- Efficient transaction verification using **Merkle Trees** for compact inclusion proofs
- Detection of and reaction to forks in the chain
- Support for multiple transactions per block
- **Coinbase transactions** that create new coins as block rewards, incentivizing honest mining

This assignment is implemented in **Python**. You may work with a partner on this program.

Algorithms and Notation.

- ||: denotes the concatenation of two bitstrings
- sha256(): denotes a sha256 hash function
- double_sha256(): denotes double sha256 hash function sha256(sha256()) attacks.

Credits for Scripts

Basics. In `script.py` these functions are the basics:

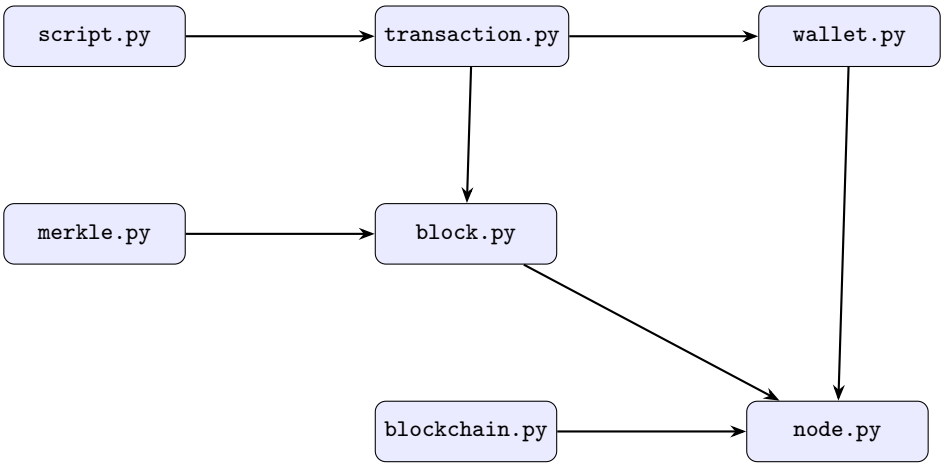
- `sha256_hash()`
- `verify_p2pkh()`

- `Script.to_bytes()`

Extra Credit. The `ScriptInterpreter` class in `script.py` is entirely extra credit. You can complete the full assignment without implementing it. The `ScriptInterpreter` implements a full stack-based script interpreter, similar to how Bitcoin actually evaluates scripts.

Suggested Implementation Order

The starter template build on each other in a layered dependency graph:



- `script.py` is the foundation—hashing and signature verification.
- `transaction.py` and `merkle.py` are independent of each other.
- `blockchain.py` is a simple container for a chain and its UTXO set.
- `node.py` is the final orchestrator that uses everything else.

2. Formats. The starter code that we provide gives a unique serialization for inputs, outputs, transactions, and blocks. Please refer to the starter code for how this is represented. For pedagogical purposes, you can envision a transaction like so:

```
{
    "number": <SHA256 hash of serialized transaction>,
    "inputs": [
        {
            "number": <transaction ID of the UTXO being spent>,
            "output": {
                "value": <value>,
                "script_pubkey": [OP_DUP, OP_HASH256, <pubKeyHash>, OP_EQUALVERIFY, OP_CHECKSIG]
            },
            "script_sig": [<signature>, <pubKey>]
        },
        ...
    ],
    "outputs": [
        {
            "value": <value>,
            "script_pubkey": [OP_DUP, OP_HASH256, <pubKeyHash>, OP_EQUALVERIFY, OP_CHECKSIG]
        },
        ...
    ]
}
```

And a block like so:

```
{
   "prev": <SHA256 hash of previous block>,
   "merkle_root": <Merkle root of all transaction hashes>,
   "nonce": <proof-of-work nonce>,
   "txs": [<transaction>, <transaction>, ...]
}
```

Note: `...` implies that there could be an arbitrary number of similar elements in the list. Nonces, signatures, and keys should be formatted as hex. We use the EdDSA (Bitcoin uses ECDSA) algorithm as implemented in `pynacl`. See `https://pynacl.readthedocs.io/en/latest/signing/` for signing and verification examples.

3. Bitcoin Script. Our implementation uses a simplified Bitcoin Script system with human-readable opcode text. Unlike real Bitcoin which uses binary opcodes, we represent scripts as lists of plaintext strings for clarity.

Note that signatures are contained within each input's `script_sig` field rather than a single transaction-level signature. The `script_pubkey` defines the spending conditions (locking script), while `script_sig` provides the data to satisfy those conditions (unlocking script). This follows Bitcoin's Pay-to-Public-Key-Hash (P2PKH) transaction format.

Pay-to-Public-Key-Hash (P2PKH) is the most common Bitcoin transaction type. It consists of two scripts:

Locking Script (scriptPubKey) is stored in the output, defines spending conditions:

```
["OP_DUP", "OP_HASH256", "<pubKeyHash>", "OP_EQUALVERIFY", "OP_CHECKSIG"]
```

Where <pubKeyHash> is the sha256(publicKey) hash of the recipient's public key, encoded as a 40-character hex string.

Unlocking Script (scriptSig) provides in the input when spending:

```
["<signature>", "<pubKey>"]
```

Where `<signature>` is the EdDSA signature (Bitcoin uses ECDSA) of the transaction data, and `<pubKey>` is the sender's public key.

Script Execution. To validate a transaction, the unlocking script is concatenated with the locking script and executed on a stack:

```
Combined: [<sig>, <pubKey>, OP_DUP, OP_HASH256, <pubKeyHash>, OP_EQUALVERIFY, OP_CHECKSIG]

Step 1: Push <sig> Stack: [sig]
Step 2: Push <pubKey> Stack: [sig, pubKey]
Step 3: OP_DUP Stack: [sig, pubKey, pubKey]
Step 4: OP_HASH256 Stack: [sig, pubKey, HASH256(pubKey)]
Step 5: Push <pubKeyHash> Stack: [sig, pubKey, HASH256(pubKey), pubKeyHash]
Step 6: OP_EQUALVERIFY Stack: [sig, pubKey] (fails if hashes don't match)
Step 7: OP_CHECKSIG Stack: [true] (fails if signature invalid)
```

The transaction is valid if the final stack contains a truthy value (non-zero, non-empty).

Opcode Definitions:

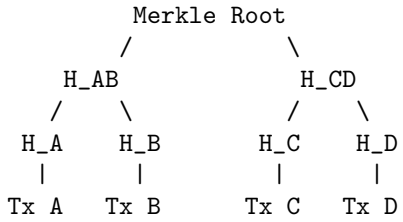
- `OP_DUP`: Duplicate the top stack element
- `OP_HASH256`: Pop the top element, push sha256(element)
- `OP_EQUALVERIFY`: Pop two elements, fail immediately if they are not equal
- `OP_CHECKSIG`: Pop pubKey and signature, verify signature against transaction data, push result (1 for valid, 0 for invalid)

4. Merkle Trees. Each block contains a Merkle root, which is a single hash that commits to all transactions in the block. This enables efficient proofs that a transaction is included in a block.

Construction Algorithm:

1. Compute the hash (transaction ID) of each transaction, producing $[H_A, H_B, H_C, H_D]$.
2. If the number of leaf nodes (or transactions) is odd, set empty nodes with zero hash 0^{256}.
3. Pair adjacent hashes and compute parent: $H_{AB} \leftarrow \mathsf{double\_sha256}(H_A \| H_B)$.
4. Repeat until only one hash remains which is the Merkle root.

Example with 4 transactions:



Hash Function: We use double_sha256() for Merkle tree nodes:

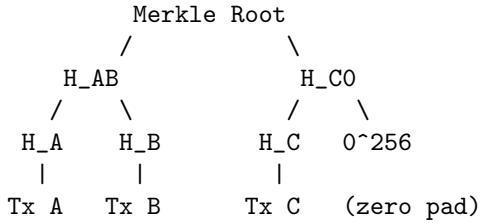
```
def double_sha256(data):
  return sha256(sha256(data).digest()).digest()

def merkle_parent(left, right):
  return double_sha256(left + right) # concatenate then hash
```

Imbalanced Trees and Zero Padding: A Merkle tree requires pairs of nodes at each level. When a block contains an odd number of transactions, the tree becomes imbalanced—one node at some level will be missing its sibling.

To handle this, we pad with a **zero hash**: 32 bytes of zeros (represented as 64 hex characters of '0'). This is defined as:

```
ZERO_HASH = '0' * 64 # 32 bytes of zeros in hexadecimal
```

Example with 3 transactions (imbalanced):



The computation proceeds as:

1. Level 0: $[H_A, H_B, H_C]$ — odd count, pad with 0^{256} to get $[H_A, H_B, H_C, 0^{256}]$
2. Level 1: $[H_{AB}, H_{C0}]$ where $H_{C0} = \mathsf{double\_sha256}(H_C \| 0^{256})$
3. Level 2: Merkle Root $= \mathsf{double\_sha256}(H_{AB} \| H_{C0})$

5. Multiple Transactions Per Block. Each block contains a list of transactions rather than a single transaction. The transactions are ordered, and this order matters:

- The **first transaction** may be a coinbase transaction (block reward)
- Subsequent transactions must be regular transactions with valid inputs
- A transaction may spend outputs created by an earlier transaction in the same block
- When validating, process transactions in order and update a temporary UTXO set

6. Hashing and Encoding. Hashes are 256 bits and should be written as 64-character hexadecimal values. You can compute these in Python as follows:

```
# python3
>>> from hashlib import sha256 as H
>>> computed_hash = H(b'hello')
>>> computed_hash.hexdigest()
'2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824'
```

7. Genesis Block. The genesis block is the first block in the blockchain and is provided by the autograder. It has special properties:

- The `prev` hash field contains arbitrary hex data (there is no previous block)
- The `nonce` and proof-of-work are pre-computed
- It contains a **coinbase transaction** that creates the initial coins

8. Coinbase Transactions. A coinbase transaction is a special transaction that creates new coins as a block reward:

- It has an **empty input list** (no UTXOs are spent)
- It does not require a signature (there are no inputs to authorize)
- Its outputs create new UTXOs that can be spent in future transactions
- The total output value must not exceed the block reward (50 coins)
- Only the **first transaction** in a block may be a coinbase transaction

The autograder will provide you with a genesis block. Your implementation should:

- Initialize the UTXO set from the genesis block's coinbase outputs
- Validate that coinbase transactions only appear as the first transaction in a block
- Reject blocks where coinbase output values exceed the block reward

9. Node behavior. Your node will receive the global genesis block from the autograder. The autograder will verify your node can create valid transactions. It will also verify that your node can build (and mine) blocks from transactions. The autograder will also verify that your node accepts valid blocks from "the network." You do not need to keep track of any pending transactions.

In order to process a transaction, a node will have to perform the following steps:

1. Ensure the transaction is validly structured
   (a) `number` hash is correct
   (b) each `input` is correct
      i. each number in the input exists as a transaction already on the blockchain
      ii. each output in the input actually exists in the named transaction
      iii. each output in the input has the same public key, and that key can verify the signature on this transaction
      iv. that public key is the most recent recipient of that output (i.e. not a double-spend)
   (c) the sum of the input and output values are equal
2. Construct a block containing this transaction, setting `prev` to the hash of the most recent block
3. Create a valid proof-of-work for this block by setting `nonce` and `pow`

Additionally, each node must react to the broadcast of blocks from "other nodes":

1. Verify the proof-of-work
2. Verify the `prev` hash
3. Validate the transaction in the block
4. If all three checks pass, append this block to the node's instance of the blockchain, and continue work
5. Handle potential forks

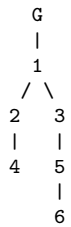
10. Proof-of-Work. In order to be a valid block, the pow value must be less than or equal to the hexadecimal value below:

```
0x07FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

11. Forking. In the operation of your nodes, forks may occur in your blockchain:

- The longest chain is the only valid blockchain
- Thus, at any point, any blocks can be invalidated, no block is ever "final"
- Your node should mine blocks based off of the longest chain it is aware of
- Spent outputs in an invalidated block become unspent transaction outputs

Forking example (numbers represent blocks in order they are broadcast, and lines represent prev links):



After 2 and 3 are broadcast (likely at roughly the same time), some nodes might be working off of 2 and some off of 3. Once 4 is broadcast, the transaction in 3 is no longer verified. It is possible (but unlikely) that blocks 5 and 6 might build off of 3 rather than 4, at which point the transactions in both 2 and 4 would be unverified – once that chain no longer represents the longest path back the Genesis block G.

12. Longest Chain Rule (Nakamoto Consensus). When building or validating blocks, always follow the longest chain:

- Always build on top of the longest valid chain
- Accept any valid block that extends any known chain tip (this creates a fork)
- When two chains have equal length, continue building on the one you saw **first** – only switch if another chain becomes strictly longer
- Forks naturally resolve as one chain grows longer; transactions in orphaned blocks return to the mempool (not implemented in this assignment)

This ensures all honest nodes eventually converge on the same chain, and an attacker would need more than 50% of the network's hash power to consistently outpace honest miners.

Output. Follow the comments in the starter code.

Notes. Running your program should require no external dependencies or packages, but installation of Python 3 packages via `pip`/`pip3` is acceptable. Include a `requirements.txt` (generated via `pip list > requirements.txt`).

Resources. You may wish to consult the following resources:

- `https://www.igvita.com/2014/05/05/minimum-viable-block-chain/`
- `https://docs.python.org/3.5/library/venv.html`
- `https://docs.python.org/3.7/library/threading.html`
- `https://en.wikibooks.org/wiki/Python_Programming/Threading`
- `https://docs.python.org/3.7/library/json.html`
- `https://docs.python.org/3/library/hashlib.html`
- `https://pynacl.readthedocs.io/en/latest/`

This project was adapted from one developed by Professor Zachary Peterson.

Submission. Submit the following 7 Python files to Gradescope (individually or as a zip): `block.py`, `blockchain.py`, `merkle.py`, `node.py`, `script.py`, `transaction.py`, and `wallet.py`. Do not include subdirectories or `__pycache__` folders. File names must match exactly.