```
================================================================================
File: __init__.py
================================================================================
# the following lines expose items defined in various files when using 'from ex2 import <item>
from .block import Block
from .transaction import Transaction
from .node import Node
from .utils import PublicKey, Signature, BlockHash, TxID, GENESIS_BLOCK_PREV, BLOCK_SIZE, sign


# this defines what to import when using 'from ex2 import *'
__all__ = ["Node", "Block", "Transaction", "PublicKey",
           "Signature", "BlockHash", "TxID", "GENESIS_BLOCK_PREV", "BLOCK_SIZE", "sign", "gen_

================================================================================
File: block.py
================================================================================
from .utils import BlockHash
from .transaction import Transaction
from typing import List


class Block:
    """This class represents a block."""

    def __init__(self, prev_block_hash: BlockHash, transactions: List[Transaction]) -> None:
        """creates a block with the given previous block hash and a list of transactions"""
        self._prev_block_hash: BlockHash = prev_block_hash
        self._transactions: List[Transaction] = list(transactions)

    def get_block_hash(self) -> BlockHash:
        """Gets the hash of this block.
        This function is used by the tests. Make sure to compute the result from the data in t
        and not to cache the result"""
        import hashlib

        hasher = hashlib.sha256()
        hasher.update(self._prev_block_hash)
        for tx in self._transactions:
            hasher.update(tx.get_txid())
        return BlockHash(hasher.digest())

    def get_transactions(self) -> List[Transaction]:
        """
        returns the list of transactions in this block.
        """
        return list(self._transactions)

    def get_prev_block_hash(self) -> BlockHash:
        """Gets the hash of the previous block"""
        return self._prev_block_hash

================================================================================
```

File: node.py

```
================================================================================
from .utils import *
from .block import Block
from .transaction import Transaction
from typing import Set, Optional, List, Dict, Tuple


class Node:
    def __init__(self) -> None:
        """Creates a new node with an empty mempool and no connections to others.
        Blocks mined by this node will reward the miner with a single new coin,
        created out of thin air and associated with the mining reward address"""
        self._private_key: PrivateKey
        self._public_key: PublicKey
        self._private_key, self._public_key = gen_keys()

        self._mempool: List[Transaction] = []
        self._mempool_inputs: Set[TxID] = set()

        self._connections: Set[Node] = set()

        self._blocks: Dict[BlockHash, Block] = {}
        self._block_heights: Dict[BlockHash, int] = {}
        self._block_utxo: Dict[BlockHash, Dict[TxID, Transaction]] = {}

        self._latest_hash: BlockHash = GENESIS_BLOCK_PREV
        self._block_heights[GENESIS_BLOCK_PREV] = 0
        self._block_utxo[GENESIS_BLOCK_PREV] = {}

    def connect(self, other: 'Node') -> None:
        """connects this node to another node for block and transaction updates.
        Connections are bi-directional, so the other node is connected to this one as well.
        Raises an exception if asked to connect to itself.
        The connection itself does not trigger updates about the mempool,
        but nodes instantly notify of their latest block to each other (see notify_of_block)""
        if other is self:
            raise ValueError("Cannot connect node to itself")
        if other not in self._connections:
            self._connections.add(other)
            other._connections.add(self)
        # exchange tip information
        other.notify_of_block(self._latest_hash, self)
        self.notify_of_block(other._latest_hash, other)

    def disconnect_from(self, other: 'Node') -> None:
        """Disconnects this node from the other node. If the two were not connected, then noth
        if other in self._connections:
            self._connections.discard(other)
            other._connections.discard(self)

    def get_connections(self) -> Set['Node']:
        """Returns a set containing the connections of this node."""
```

```
        return set(self._connections)

    def add_transaction_to_mempool(self, transaction: Transaction) -> bool:
        """
        This function inserts the given transaction to the mempool.
        It will return False iff any of the following conditions hold:
        (i) the transaction is invalid (the signature fails)
        (ii) the source doesn't have the coin that it tries to spend
        (iii) there is contradicting tx in the mempool.

        If the transaction is added successfully, then it is also sent to neighboring nodes.
        Transactions that create money (with no inputs) are not placed in the mempool, and not
        """
        # coinbase txs are not accepted to the mempool
        if transaction.input is None:
            return False

        # already in mempool
        if transaction in self._mempool:
            return False

        # check double-spend in mempool
        if transaction.input in self._mempool_inputs:
            return False

        prev_tx = self._block_utxo[self._latest_hash].get(transaction.input)
        if prev_tx is None:
            return False

        message = self._serialize_for_signature(transaction.input, transaction.output)
        if not verify(message, transaction.signature, prev_tx.output):
            return False

        self._mempool.append(transaction)
        self._mempool_inputs.add(transaction.input)

        # propagate to neighbors
        for neighbor in list(self._connections):
            neighbor.add_transaction_to_mempool(transaction)
        return True

    def notify_of_block(self, block_hash: BlockHash, sender: 'Node') -> None:
        """This method is used by a node's connection to inform it that it has learned of a
        new block (or created a new block). If the block is unknown to the current Node, The b
        We assume the sender of the message is specified, so that the node can choose to reque
        it wishes to do so.
        (if it is part of a longer unknown chain, these blocks are requested as well, until re
        Upon receiving new blocks, they are processed and and checked for validity (check all
        block size , etc).
        If the block is on the longest chain, the mempool and utxo change accordingly (ties, i
        If the block is indeed the tip of the longest chain,
        a notification of this block is sent to the neighboring nodes of this node.
        (no need to notify of previous blocks -- the nodes will fetch them if needed)
```

```
    A reorg may be triggered by this block's introduction. In this case the utxo is rolled
    and then rolled forward along the new branch. Be careful -- the new branch may contain
    the mempool is similarly emptied of transactions that cannot be executed now.
    transactions that were rolled back and can still be executed are re-introduced into th
    not conflict.
    """
    # If already known, we may still need to update best tip
    if block_hash not in self._blocks and block_hash != GENESIS_BLOCK_PREV:
        self._fetch_and_store_chain(block_hash, sender)

    # update best chain if needed
    self._adopt_best_chain_if_needed()

    # notify neighbors if new tip is this block
    if self._latest_hash == block_hash:
        for neighbor in list(self._connections):
            if neighbor is not sender:
                neighbor.notify_of_block(block_hash, self)

def mine_block(self) -> BlockHash:
    """"
    This function allows the node to create a single block.
    The block should contain BLOCK_SIZE transactions (unless there aren't enough in the me
    BLOCK_SIZE-1 transactions come from the mempool and one addtional transaction will be
    money and adds it to the address of this miner.
    Money creation transactions have None as their input, and instead of a signature, cont
    If a new block is created, all connections of this node are notified by calling their
    The method returns the new block hash.
    """
    import secrets

    # select transactions from mempool (excluding potential double spends which are alread
    usable_txs = self._mempool[: max(0, BLOCK_SIZE - 1)]

    coinbase_sig = Signature(secrets.token_bytes(64))
    coinbase_tx = Transaction(output=self._public_key, tx_input=None, signature=coinbase_s

    block_txs = [coinbase_tx] + usable_txs
    block = Block(self._latest_hash, block_txs)
    block_hash = block.get_block_hash()

    # Validate and store as if received externally
    if self._store_block(block, self._latest_hash):
        self._adopt_best_chain_if_needed()
        # clear spent txs from mempool (those included)
        for tx in usable_txs:
            if tx in self._mempool:
                self._mempool.remove(tx)
            self._mempool_inputs.discard(tx.input)
        # notify neighbors
        for neighbor in list(self._connections):
            neighbor.notify_of_block(block_hash, self)
```

```
            return block_hash

        raise ValueError("Failed to mine block")

    def get_block(self, block_hash: BlockHash) -> Block:
        """
        This function returns a block object given its hash.
        If the block doesnt exist, a ValueError is raised.
        """
        if block_hash == GENESIS_BLOCK_PREV:
            raise ValueError("Genesis placeholder has no block")
        if block_hash not in self._blocks:
            raise ValueError("Block not found")
        return self._blocks[block_hash]

    def get_latest_hash(self) -> BlockHash:
        """
        This function returns the last block hash known to this node (the tip of its current c
        """
        return self._latest_hash

    def get_mempool(self) -> List[Transaction]:
        """
        This function returns the list of transactions that didn't enter any block yet.
        """
        return list(self._mempool)

    def get_utxo(self) -> List[Transaction]:
        """
        This function returns the list of unspent transactions.
        """
        return list(self._block_utxo[self._latest_hash].values())

    # ------------ Formerly wallet methods: -----------------------

    def create_transaction(self, target: PublicKey) -> Optional[Transaction]:
        """
        This function returns a signed transaction that moves an unspent coin to the target.
        It chooses the coin based on the unspent coins that this node has.
        If the node already tried to spend a specific coin, and such a transaction exists in i
        but it did not yet get into the blockchain then it should'nt try to spend it again (un
        called -- which will wipe the mempool and thus allow to attempt these re-spends).
        The method returns None if there are no outputs that have not been spent already.

        The transaction is added to the mempool (and as a result is also published to neighbor
        """
        tx = self._create_signed_transaction(target)
        if tx is None:
            return None
        added = self.add_transaction_to_mempool(tx)
        return tx if added else None

    def clear_mempool(self) -> None:
```

```
    """
    Clears the mempool of this node. All transactions waiting to be entered into the next
    """
    self._mempool.clear()
    self._mempool_inputs.clear()

def get_balance(self) -> int:
    """
    This function returns the number of coins that this node owns according to its view of
    Coins that the node owned and sent away will still be considered as part of the balanc
    transaction is in the blockchain.
    """
    return len([tx for tx in self._block_utxo[self._latest_hash].values() if tx.output ==

def get_address(self) -> PublicKey:
    """
    This function returns the public address of this node (its public key).
    """
    return self._public_key

# ----------------------------------------------------------------
# Internal helpers

@staticmethod
def _serialize_for_signature(txid: TxID, target: PublicKey) -> bytes:
    return txid + target

def _create_signed_transaction(self, target: PublicKey) -> Optional[Transaction]:
    # choose first available UTXO not already in mempool
    utxo = self._block_utxo[self._latest_hash]
    for txid, tx in utxo.items():
        if txid not in self._mempool_inputs and tx.output == self._public_key:
            message = self._serialize_for_signature(txid, target)
            signature = sign(message, self._private_key)
            return Transaction(output=target, tx_input=txid, signature=signature)
    return None

def _fetch_and_store_chain(self, block_hash: BlockHash, sender: 'Node') -> None:
    """Fetch unknown blocks from sender back to known ancestor and store if valid."""
    pending: List[Tuple[BlockHash, Block]] = []
    current_hash = block_hash
    # walk back until known or genesis placeholder
    while current_hash not in self._blocks and current_hash != GENESIS_BLOCK_PREV:
        try:
            block = sender.get_block(current_hash)
        except Exception:
            return
        # refuse blocks whose hash does not match the requested hash
        if block.get_block_hash() != current_hash:
            return
        pending.append((current_hash, block))
        current_hash = block.get_prev_block_hash()
```

```
    # process from ancestor forward
    for h, blk in reversed(pending):
        parent_hash = blk.get_prev_block_hash()
        if parent_hash not in self._block_utxo:
            # parent invalid / unknown
            continue
        if not self._store_block(blk, parent_hash):
            continue

def _store_block(self, block: Block, parent_hash: BlockHash) -> bool:
    """Validate block given known parent utxo and store state."""
    if parent_hash not in self._block_utxo:
        return False
    if block.get_prev_block_hash() != parent_hash:
        return False

    txs = block.get_transactions()
    # size check
    if len(txs) > BLOCK_SIZE or len(txs) == 0:
        return False

    # exactly one coinbase
    coinbase_count = sum(1 for tx in txs if tx.input is None)
    if coinbase_count != 1:
        return False

    parent_utxo = self._block_utxo[parent_hash]
    new_utxo: Dict[TxID, Transaction] = dict(parent_utxo)

    spent_inputs: Set[TxID] = set()
    for tx in txs:
        if tx.input is None:
            # money creation
            pass
        else:
            # must exist and not double spend
            if tx.input in spent_inputs:
                return False
            prev_tx = new_utxo.get(tx.input)
            if prev_tx is None:
                return False
            message = self._serialize_for_signature(tx.input, tx.output)
            if not verify(message, tx.signature, prev_tx.output):
                return False
            spent_inputs.add(tx.input)
            new_utxo.pop(tx.input, None)
        # add new output
        new_utxo[tx.get_txid()] = tx

    block_hash = block.get_block_hash()
    # store
    self._blocks[block_hash] = block
    height = self._block_heights.get(parent_hash, 0) + 1
```

```
        self._block_heights[block_hash] = height
        self._block_utxo[block_hash] = new_utxo
        return True

    def _adopt_best_chain_if_needed(self) -> None:
        """Pick the longest known chain (highest height). Ties keep current."""
        best_hash = self._latest_hash
        best_height = self._block_heights.get(best_hash, 0)

        for h, height in self._block_heights.items():
            if height > best_height:
                best_height = height
                best_hash = h

        if best_hash != self._latest_hash:
            self._latest_hash = best_hash
            self._refresh_mempool_after_reorg()

    def _refresh_mempool_after_reorg(self) -> None:
        """Drop invalid mempool txs and reinsert those still valid under new UTXO."""
        current_utxo = self._block_utxo[self._latest_hash]
        new_mempool: List[Transaction] = []
        new_inputs: Set[TxID] = set()

        for tx in self._mempool:
            if tx.input is None:
                continue
            if tx.input in new_inputs:
                continue
            prev_tx = current_utxo.get(tx.input)
            if prev_tx is None:
                continue
            message = self._serialize_for_signature(tx.input, tx.output)
            if not verify(message, tx.signature, prev_tx.output):
                continue
            new_mempool.append(tx)
            new_inputs.add(tx.input)

        self._mempool = new_mempool
        self._mempool_inputs = new_inputs


"""
Importing this file should NOT execute code. It should only create definitions for the objects
Write any tests you have in a different file.
You may add additional methods, classes and files but be sure no to change the signatures of m
included in this template.
"""

================================================================================
File: transaction.py
================================================================================
from .utils import PublicKey, Signature, TxID
```

```
from typing import Optional


class Transaction:
    """Represents a transaction that moves a single coin
    A transaction with no source creates money. It will only be created by the miner of a bloc

    def __init__(self, output: PublicKey, tx_input: Optional[TxID], signature: Signature) -> N
        # DO NOT change these field names.
        self.output: PublicKey = output
        # DO NOT change these field names.
        self.input: Optional[TxID] = tx_input
        # DO NOT change these field names.
        self.signature: Signature = signature

    def get_txid(self) -> TxID:
        """
        Returns the identifier of this transaction. This is the sha256 of the transaction cont
        This function is used by the tests to compute the tx hash. Make sure to compute this e
        directly from the data in the transaction object, and not cache the result
        """
        import hashlib

        hasher = hashlib.sha256()
        if self.input is None:
            hasher.update(b"\x00")
        else:
            hasher.update(b"\x01")
            hasher.update(self.input)
        hasher.update(self.output)
        hasher.update(self.signature)
        return TxID(hasher.digest())


"""
Importing this file should NOT execute code. It should only create definitions for the objects
Write any tests you have in a different file.
You may add additional methods, classes and files but be sure no to change the signatures of m
included in this template.
"""

================================================================================
File: utils.py
================================================================================
from cryptography.hazmat.primitives.serialization import Encoding, PublicFormat, PrivateFormat
from cryptography.hazmat.primitives.asymmetric.ed25519 import Ed25519PrivateKey, Ed25519Public
from typing import NewType, Tuple

# The following types are used to distinguish between bytes that are used as private keys, pub
# This utilizes typechecking to ensure we won't be using them interchangeably.
PrivateKey = NewType('PrivateKey', bytes)
PublicKey = NewType('PublicKey', bytes)
Signature = NewType('Signature', bytes)
```

```
# We make similar type definitions for hashes:
BlockHash = NewType('BlockHash', bytes)  # This will be the hash of a block
TxID = NewType("TxID", bytes)  # this will be a hash of a transaction

# these are the bytes written as the prev_block_hash of the 1st block.
# (when a new wallet is created, it is updated up to this point)
GENESIS_BLOCK_PREV = BlockHash(b"Genesis")
# The maximal size of a block. Larger blocks are illegal. Do not change this value.
BLOCK_SIZE = 10


def sign(message: bytes, private_key: PrivateKey) -> Signature:
    """Signs the given message using the given private key"""
    pk = Ed25519PrivateKey.from_private_bytes(
        private_key)
    return Signature(pk.sign(message))


def verify(message: bytes, sig: Signature, pub_key: PublicKey) -> bool:
    """Verifies a signature for a given message using a public key.
    Returns True is the signature matches, otherwise False"""
    pub_k = Ed25519PublicKey.from_public_bytes(
        pub_key)
    try:
        pub_k.verify(sig, message)
        return True
    except:
        return False


def gen_keys() -> Tuple[PrivateKey, PublicKey]:
    """generates a private key and a corresponding public key.
    The keys are returned in byte format to allow them to be serialized easily."""
    private_key = Ed25519PrivateKey.generate()
    priv_key_bytes = private_key.private_bytes(
        Encoding.Raw, PrivateFormat.Raw, encryption_algorithm=NoEncryption())
    pub_key_bytes = private_key.public_key().public_bytes(
        Encoding.Raw, PublicFormat.Raw)
    return PrivateKey(priv_key_bytes), PublicKey(pub_key_bytes)

================================================================================
File: requirements.txt
================================================================================
mypy
pytest
cryptography
```