```
import hashlib, json, sys

def hashMe(msg=""):
    # For convenience, this is a helper function that wraps our hashing algorithm
    if type(msg)!=str:
        msg = json.dumps(msg,sort_keys=True)  # If we don't sort keys, we can't guarantee repeatability!

    if sys.version_info.major == 2:
        return unicode(hashlib.sha256(msg).hexdigest(),'utf-8')
    else:
        return hashlib.sha256(str(msg).encode('utf-8')).hexdigest()


import random
random.seed(0)

def makeTransaction(maxValue=3):
    # This will create valid transactions in the range of (1,maxValue)
    sign      = int(random.getrandbits(1))*2 - 1   # This will randomly choose -1 or 1
    amount    = random.randint(1,maxValue)
    alicePays = sign * amount
    bobPays   = -1 * alicePays
    # By construction, this will always return transactions that respect the conservation of tokens.
    # However, note that we have not done anything to check whether these overdraft an account
    return {u'Alice':alicePays,u'Bob':bobPays}


txnBuffer = [makeTransaction() for i in range(30)]


def updateState(txn, state):
    # Inputs: txn, state: dictionaries keyed with account names, holding numeric values for transfer amount (txn) or account balance (state)
    # Returns: Updated state, with additional users added to state if necessary
    # NOTE: This does not not validate the transaction- just updates the state!

    # If the transaction is valid, then update the state
    state = state.copy() # As dictionaries are mutable, let's avoid any confusion by creating a working copy of the data.
    for key in txn:
        if key in state.keys():
            state[key] += txn[key]
        else:
            state[key] = txn[key]
    return state


def isValidTxn(txn,state):
    # Assume that the transaction is a dictionary keyed by account names

    # Check that the sum of the deposits and withdrawals is 0
    if sum(txn.values()) is not 0:
        return False

    # Check that the transaction does not cause an overdraft
    for key in txn.keys():
        if key in state.keys():
            acctBalance = state[key]
        else:
            acctBalance = 0
        if (acctBalance + txn[key]) < 0:
            return False

    return True
```

```
<>:5: SyntaxWarning: "is not" with a literal. Did you mean "!="?
<>:5: SyntaxWarning: "is not" with a literal. Did you mean "!="?
<ipython-input-5-ed27507b87a7>:5: SyntaxWarning: "is not" with a literal. Did you mean "!="?
  if sum(txn.values()) is not 0:
```

```
state = {u'Alice':5,u'Bob':5}

print(isValidTxn({u'Alice': -3, u'Bob': 3},state))  # Basic transaction- this works great!
print(isValidTxn({u'Alice': -4, u'Bob': 3},state))  # But we can't create or destroy tokens!
print(isValidTxn({u'Alice': -6, u'Bob': 6},state))  # We also can't overdraft our account.
print(isValidTxn({u'Alice': -4, u'Bob': 2,'Lisa':2},state)) # Creating new users is valid
print(isValidTxn({u'Alice': -4, u'Bob': 3,'Lisa':2},state)) # But the same rules still apply!
```

```
True
False
False
True
False
```

```
state = {u'Alice':50, u'Bob':50}  # Define the initial state
genesisBlockTxns = [state]
genesisBlockContents = {u'blockNumber':0,u'parentHash':None,u'txnCount':1,u'txns':genesisBlockTxns}
genesisHash = hashMe( genesisBlockContents )
genesisBlock = {u'hash':genesisHash,u'contents':genesisBlockContents}
genesisBlockStr = json.dumps(genesisBlock, sort_keys=True)


chain = [genesisBlock]



def makeBlock(txns,chain):
    parentBlock = chain[-1]
    parentHash  = parentBlock[u'hash']
    blockNumber = parentBlock[u'contents'][u'blockNumber'] + 1
    txnCount    = len(txns)
    blockContents = {u'blockNumber':blockNumber,u'parentHash':parentHash,
                     u'txnCount':len(txns),'txns':txns}
    blockHash = hashMe( blockContents )
    block = {u'hash':blockHash,u'contents':blockContents}

    return block


blockSizeLimit = 5  # Arbitrary number of transactions per block-
               #  this is chosen by the block miner, and can vary between blocks!

while len(txnBuffer) > 0:
    bufferStartSize = len(txnBuffer)

    ## Gather a set of valid transactions for inclusion
    txnList = []
    while (len(txnBuffer) > 0) & (len(txnList) < blockSizeLimit):
        newTxn = txnBuffer.pop()
        validTxn = isValidTxn(newTxn,state) # This will return False if txn is invalid

        if validTxn:           # If we got a valid state, not 'False'
            txnList.append(newTxn)
            state = updateState(newTxn,state)
        else:
            print("ignored transaction")
            sys.stdout.flush()
            continue  # This was an invalid transaction; ignore it and move on

    ## Make a block
    myBlock = makeBlock(txnList,chain)
    chain.append(myBlock)


chain[0]
```

```
{'hash': '7c88a4312054f89a2b73b04989cd9b9e1ae437e1048f89fbb4e18a08479de507',
 'contents': {'blockNumber': 0,
  'parentHash': None,
  'txnCount': 1,
  'txns': [{'Alice': 50, 'Bob': 50}]}}
```

```
chain[1]
```

```
{'hash': '7a91fc8206c5351293fd11200b33b7192e87fad6545504068a51aba868bc6f72',
 'contents': {'blockNumber': 1,
  'parentHash': '7c88a4312054f89a2b73b04989cd9b9e1ae437e1048f89fbb4e18a08479de507',
  'txnCount': 5,
  'txns': [{'Alice': 3, 'Bob': -3},
   {'Alice': -1, 'Bob': 1},
   {'Alice': 3, 'Bob': -3},
   {'Alice': -2, 'Bob': 2},
   {'Alice': 3, 'Bob': -3}]}}
```

```
state
```

```
{'Alice': 72, 'Bob': 28}
```

```
def checkBlockHash(block):
    # Raise an exception if the hash does not match the block contents
    expectedHash = hashMe( block['contents'] )
    if block['hash']!=expectedHash:
        raise Exception('Hash does not match contents of block %s'%
                        block['contents']['blockNumber'])
    return


def checkBlockValidity(block,parent,state):
    # We want to check the following conditions:
    # - Each of the transactions are valid updates to the system state
    # - Block hash is valid for the block contents
    # - Block number increments the parent block number by 1
    # - Accurately references the parent block's hash
    parentNumber = parent['contents']['blockNumber']
    parentHash   = parent['hash']
    blockNumber  = block['contents']['blockNumber']

    # Check transaction validity; throw an error if an invalid transaction was found.
    for txn in block['contents']['txns']:
        if isValidTxn(txn,state):
            state = updateState(txn,state)
        else:
            raise Exception('Invalid transaction in block %s: %s'%(blockNumber,txn))

    checkBlockHash(block) # Check hash integrity; raises error if inaccurate

    if blockNumber!=(parentNumber+1):
        raise Exception('Hash does not match contents of block %s'%blockNumber)

    if block['contents']['parentHash'] != parentHash:
        raise Exception('Parent hash not accurate at block %s'%blockNumber)

    return state


def checkChain(chain):
    # Work through the chain from the genesis block (which gets special treatment),
    #  checking that all transactions are internally valid,
    #    that the transactions do not cause an overdraft,
    #    and that the blocks are linked by their hashes.
    # This returns the state as a dictionary of accounts and balances,
    #   or returns False if an error was detected


    ## Data input processing: Make sure that our chain is a list of dicts
    if type(chain)==str:
        try:
            chain = json.loads(chain)
            assert( type(chain)==list)
        except:  # This is a catch-all, admittedly crude
            return False
    elif type(chain)!=list:
        return False

    state = {}
    ## Prime the pump by checking the genesis block
    # We want to check the following conditions:
    # - Each of the transactions are valid updates to the system state
    # - Block hash is valid for the block contents

    for txn in chain[0]['contents']['txns']:
        state = updateState(txn,state)
    checkBlockHash(chain[0])
    parent = chain[0]

    ## Checking subsequent blocks: These additionally need to check
    #    - the reference to the parent block's hash
    #    - the validity of the block number
    for block in chain[1:]:
        state = checkBlockValidity(block,parent,state)
        parent = block

    return state
```

```
checkChain(chain)
```

```
{'Alice': 72, 'Bob': 28}
```

```
chainAsText = json.dumps(chain,sort_keys=True)
checkChain(chainAsText)
```

```
{'Alice': 72, 'Bob': 28}
```

```
import copy
nodeBchain = copy.copy(chain)
nodeBtxns  = [makeTransaction() for i in range(5)]
newBlock   = makeBlock(nodeBtxns,nodeBchain)
```

```
print("Blockchain on Node A is currently %s blocks long"%len(chain))

try:
    print("New Block Received; checking validity...")
    state = checkBlockValidity(newBlock,chain[-1],state) # Update the state- this will throw an error if the block is invalid!
    chain.append(newBlock)
except:
    print("Invalid block; ignoring and waiting for the next block...")

print("Blockchain on Node A is now %s blocks long"%len(chain))
```

```
Blockchain on Node A is currently 7 blocks long
New Block Received; checking validity...
Blockchain on Node A is now 8 blocks long
```

Start coding or generate with AI.