```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives.asymmetric import padding

def generate_key(alias, password):
    # Generate the private/public key pair with the password.
    private_key = rsa.generate_private_key(
      public_exponent = 65537,
      key_size = 2048,
      backend = default_backend(),
    )

    # Serialize the private key
    pem_private =  private_key.private_bytes(
      encoding=serialization.Encoding.PEM,
      format=serialization.PrivateFormat.TraditionalOpenSSL,
      encryption_algorithm=serialization.BestAvailableEncryption(password),
     )

    # Save the private key to a file 'private_key'.
    with open('private_key_of_' + alias, 'wb') as f:
      f.write(pem_private)
      f.close()

    # Serialize the public key
    pem_public = private_key.public_key().public_bytes(
       encoding = serialization.Encoding.PEM,
       format = serialization.PublicFormat.SubjectPublicKeyInfo,
       )

    # Save the public key to a file 'public_key'.
    with open('public_key_of_' + alias, 'wb') as f:
      f.write(pem_public)
      f.close()


def sign(trans, alias, password):
    # Load the private key.
    with open('private_key_of_' + alias, 'rb') as f:
      private_key = serialization.load_pem_private_key(
            f.read(),
            password = password,
            backend = default_backend(),
      )
      f.close()
```

```
    # Sign the transaction.
    signature = private_key.sign(
      trans,
      padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
                  salt_length=padding.PSS.MAX_LENGTH
      ),
      hashes.SHA256()
     )

    return signature

def verify_sig(transaction, sign, alias):
    # Load the public key.
    with open('public_key_of_' + alias, 'rb') as f:
      public_key = serialization.load_pem_public_key(f.read(),
default_backend())
      f.close()

    # Perform the verification.
    try:
       public_key.verify(
       sign,
       transaction,
          padding.PSS(
             mgf = padding.MGF1(hashes.SHA256()),
             salt_length = padding.PSS.MAX_LENGTH,
          ),
          hashes.SHA256(),
    )
       return True
    except cryptography.exceptions.InvalidSignature as e:
      return False
```

Goofycoin

```
import random
import time
import hashlib
from crypto_key import generate_key
```

```
from crypto_key import sign
from crypto_key import verify_sig

class Coin:
    def __init__(self, alias, password):
      n = random.randint(0, 10000)
      self._coin_id = "CreateCoin_" + str(n)
        self._time_stamp = time.time()
        self._sig = sign(self.digest(), alias, password)

    def digest(self):
        m = hashlib.sha256()
        m.update(
            str(self._coin_id).encode('utf-8') +
            str(self._time_stamp).encode('utf-8')
            )
        return m.hexdigest()

class Transaction:
    def __init__(self, sender, coin, receiver, password):
      self._trans_id = sender + " tranfer " + str(coin) + " to " + receiver
      self._sender = sender
      self._receiver = receiver
      self._coin = coin
        self._time_stamp = time.time()
        self._sig = sign(self.digest(), sender, password)

    def digest(self):
        m = hashlib.sha256()
        m.update(
            str(self._trans_id).encode('utf-8') +
            str(self._time_stamp).encode('utf-8')
            )
        return m.hexdigest()


class Goofy:
    def __init__(self, alias, password):
      self.__alias = alias
      self.__password = password
        self.__key = generate_key(alias, password)

    def create_coin(self):
      c = Coin(self.__alias, self.__password)
      return c


def verify_coin(coin, alias, goofy):
```

```
      if isinstance(coin, Coin):
            return verify_sig(coin.digest(), coin._sig, goofy)
      if isinstance(coin, Transaction):
            return (coin._receiver == alias) and verify_sig(coin.digest(),
coin._sig, coin._sender) and verify_coin(coin._coin, coin._sender, goofy)
      return False


# TEST
g = Goofy("goofy", "g")

#1. Goofy creates three coins
coin_1 = g.create_coin()
coin_2 = g.create_coin()
coin_3 = g.create_coin()

#2. Generate three new account Alice, Bob and Celeb
Alice = generate_key("alice", "a")
Bob = generate_key("bob", "b")
Caleb = generate_key("caleb", "c")

#3. Goofy transfers one coin to Alice
trans_goofy_alice = Transaction("goofy", coin_1, "alice", "g")

#4. Goofy transfers two coins to Bob
trans_goofy_bob_1 = Transaction("goofy", coin_2, "bob", "g")
trans_goofy_bob_2 = Transaction("goofy", coin_3, "bob", "g")

#5. Alice transfers Alice coin to Bob
trans_alice_bob = Transaction("alice", trans_goofy_alice, "bob", "a")

#6. Bob transfers one of Bobs coins to Caleb
trans_bob_caleb = Transaction("bob", trans_goofy_bob_1, "caleb", "b")

#Alice transfers Alice coin to Caleb
trans_alice_caleb = Transaction("alice", trans_goofy_alice, "caleb", "a")


#Verify the transaction [Alice] transfer [coin] to [Bob], in which [coin] is
the hash pointers points  to the transaction [Bob] transfer [coin] to
[Celeb] in (6)
trans_alice_bob_f = Transaction("alice", trans_bob_caleb, "bob", "a")

print verify_coin(coin_1, "goofy", "goofy")
#True
print verify_coin(coin_2, "goofy", "goofy")
#True
print verify_coin(coin_3, "goofy", "goofy")
```

```
#True

print verify_coin(trans_goofy_alice, "alice", "goofy")
#True
print verify_coin(trans_goofy_bob_1, "bob", "goofy")
#True
print verify_coin(trans_goofy_bob_2, "bob", "goofy")
#True

print verify_coin(trans_alice_bob, "bob", "goofy")
#True

#8. Verify whether all Caleb coins are valid
print verify_coin(trans_bob_caleb, "caleb", "goofy")
#True
print verify_coin(trans_alice_caleb, "caleb", "goofy")
#True

# Verify the transaction [Alice] transfer [coin] to [Bob]
print verify_coin(trans_alice_bob_f, "bob", "goofy")
#False

#Just more tests: (coin_2) goofy -> bob -> caleb -> alice
trans_caleb_alice = Transaction("caleb", trans_bob_caleb, "alice", "c")
print verify_coin(trans_caleb_alice, "alice", "goofy")

print trans_goofy_bob_1._trans_id
```

Scroogecoin

```
import hashlib
from crypto_key import generate_key
from crypto_key import sign
from crypto_key import verify_sig

class Transaction:
```

```
    def __init__(self, data):
      self._data = data

    def sign(self, alias, password):
      self._data["signatures"].append(sign(self.digest(), alias, password))

    def digest(self):
        m = hashlib.sha256()
      if self._data["type"] == "PayCoins":
                  m.update (str(self._data["type"]))
                  m.update (str(self._data["coins_consumed"]))
            m.update (str(self._data["coins_created"]))
      else:
            m.update (str(self._data["type"]))
            m.update (str(self._data["coins_created"]))
      return bytearray(m.hexdigest(), 'utf-8')

    def get_value_alias(self, coin_id):
      for e in self._data["coins_created"]:
            if e["num"] == coin_id:
                  return [e["value"], e["recipient"]]
      return 0

    def transfer_coin_id_list(self):
      coin_list = self._data["coins_consumed"]
        arr = []
      for i in range (0,len (coin_list), 1):
            arr.append(coin_list[i].split("."))
      return arr

    def get_sum_coin_create(self):
      sum = 0
      for e in self._data["coins_created"]:
            sum = sum + e["value"]
      return sum

    def verify_sig_trans(self, alias):
      content = self.digest()
      for e in self._data["signatures"]:
            if verify_sig(content, e, alias):
                  return True
      return False


class Block:
    def __init__ (self, trans, alias, password, prev = None):
        self._prev = prev
        self._id = prev._id + 1 if prev else 0
```

```
      trans._data["transID"] = self._id
        self._trans = trans
        self._prev_hash = prev.digest() if prev is not None else
bytearray(256)
      self._sig = sign(self.digest(), alias, password)

    def digest(self):
        m = hashlib.sha256()
        m.update (str(self._id))
      m.update (str(self._trans.digest()))
      m.update (str(self._prev_hash))
        return bytearray(m.hexdigest(), 'utf-8')

    def verify(self, root_hash):
        my_hash = self.digest()
        if (root_hash != my_hash):
            print ("Hash does not verify for block containing",
self._trans._data)
        return (root_hash == my_hash
                    and (not self._prev or self._prev.verify
(self._prev_hash)))


class Scroogecoin:
    def __init__(self, alias, password):
        self._root_hash = bytearray (256)
        self._head = None
      self._alias = alias
      self.__password = password
        self.__key = generate_key(alias, password)

    def create_coin(self, data):
      self.add_transaction(data)

    def add_block(self, trans):
        new_block = Block (trans, self._alias, self.__password, self._head)
        self._root_hash = new_block.digest()
        self._head = new_block

    def add_transaction(self, trans):
      if trans._data["type"] == "CreateCoins":
            self.add_block(trans)
      else:
            if trans._data["type"] == "PayCoins":
                  if self.verify_trans(trans):
                        self.add_block(trans)
                  else:
                        print "The transaction is not valid"
```

```
            else:
                  print "Unknown transaction"

    def verify(self):
        return not self._head or self._head.verify (self._root_hash)

    def get_trans(self, trans_id, coin_id):
        pointer = self._head
      while pointer:
            if (pointer._id == trans_id):
                  return pointer._trans.get_value_alias(coin_id)
            else:
                  pointer = pointer._prev
      return None

    def check_double_spending(self, comsumed):
      pointer = self._head
      while pointer:
            for e in comsumed:
                  if pointer._trans._data["type"] == "PayCoins":
                        if (e in pointer._trans._data["coins_consumed"]):
                              return True
            pointer = pointer._prev
      return False


    def verify_trans(self, trans):
      if self.check_double_spending(trans._data["coins_consumed"]):
            print "Double spending"
            return False
      sum_coin = 0
      coins_consumed = trans.transfer_coin_id_list()
      for e in coins_consumed:
            p = self.get_trans(int (e[0]), int (e[1]))
            if p == None:
                  return False
            else:
                  sum_coin = sum_coin + float (p[0])
                  if not trans.verify_sig_trans(p[1]):
                        print "the signature is not valid"
                        return False
      return sum_coin >= trans.get_sum_coin_create()


#Scrooge Coin
scr = Scroogecoin("scrooge", "s")

#1. Generate three new account Alice, Bob, Caleb and marry
```

```
Alice = generate_key("alice", "a")
Bob = generate_key("bob", "b")
Caleb = generate_key("caleb", "c")
Marry = generate_key("marry", "m")

#2. Scrooge create new coins
json_1 = {"transID": 0,
 "type": "CreateCoins",
 "coins_created": [
  {"num": 0, "value": 3.2, "recipient": "alice"},
  {"num": 1, "value": 1.4, "recipient": "bob"},
  {"num": 2, "value": 7.5, "recipient": "caleb"}
 ]}
coin = Transaction(json_1)
scr.create_coin(coin)
print scr._head._trans._data

#3. Alice and Bob transfer their coins to Caleb and Marry: 2.5 to Caleb and
2.1 to Marry
json_2 = {"transID": 1,
"type": "PayCoins",
"coins_consumed": ["0.0", "0.1"],
"coins_created": [
{"num": 0, "value": 2.5, "recipient": "caleb"},
{"num": 1, "value": 2.1, "recipient": "marry"},
],
"signatures" : []}
trans_2 = Transaction(json_2)
trans_2.sign("alice", "a")
trans_2.sign("bob", "b")
scr.add_transaction(trans_2)
print scr._head._trans._data

#4. Caleb transfer 4.0 coins to Bob and 2.0 coins to Marry
json_3 = {"transID": 2,
"type": "PayCoins",
"coins_consumed": ["0.2", "1.0"],
"coins_created": [
{"num": 0, "value": 4.0, "recipient": "bob"},
{"num": 1, "value": 2.0, "recipient": "marry"},
{"num": 2, "value": 4.0, "recipient": "caleb"},
],
"signatures" : []}
trans_3 = Transaction(json_3)
trans_3.sign("caleb", "c")
scr.add_transaction(trans_3)
print scr._head._trans._data
```

```
#5. Alice transfer  3.2 to Bob (Double spending)

json_4 = {"transID": 3,
"type": "PayCoins",
"coins_consumed": ["0.0"],
"coins_created": [
{"num": 0, "value": 3.2, "recipient": "bob"},
],
"signatures" : []}
trans_4 = Transaction(json_4)
trans_4.sign("alice", "a")
scr.add_transaction(trans_4)
print scr._head._trans._data


#6. Alice and Bob transfer 4.0 coins to Caleb (Double spending)
json_5 = {"transID": 4,
"type": "PayCoins",
"coins_consumed": ["0.0", "0.1"],
"coins_created": [
{"num": 0, "value": 4.0, "recipient": "caleb"},
],
"signatures" : []}
trans_5 = Transaction(json_5)
trans_5.sign("alice", "a")
trans_5.sign("bob", "b")
scr.add_transaction(trans_5)
print scr._head._trans._data


#7. Verify whether the  blockchain is valid
print scr.verify()

#8. Create a new block that includes the transaction Bob transfer 2.0 to
Alice, sign this block by Alice and includes the block to the blockchain
json_6 = {"transID": 4,
"type": "PayCoins",
"coins_consumed": ["2.0"],
"coins_created": [
{"num": 0, "value": 2.0, "recipient": "alice"},
],
"signatures" : []}
trans_6 = Transaction(json_6)
trans_6.sign("alice", "a")
scr.add_transaction(trans_6)
print scr._head._trans._data

#9. Temple the first block
```

```
def get_first_block(s):
        pointer = s._head
      while pointer._prev:
                  pointer = pointer._prev
      return pointer
first_block = get_first_block(scr)
print first_block._trans._data
first_block._trans._data["coins_created"] = [
  {"num": 0, "value": 5.0, "recipient": "alice"},
  {"num": 1, "value": 1.4, "recipient": "bob"},
  {"num": 2, "value": 7.5, "recipient": "caleb"}
 ]
print first_block._trans._data
print scr.verify()

l = ["4.0", "3.1"]
print scr.check_double_spending(l)
```