

用Python实现简单的区块链系统

本篇内容对从技术角度来解释区块链，用Python实现简单的区块链系统，进一步认识区块链的结构与原理。

1、前提条件

- 需要了解区块链的概念。

区块链的概念起源于比特币，其本质上是一个去中心化的数据库，是分布式数据存储、点对点传输（P2P）、共识机制、加密算法等计算机技术的新型应用模式（摘自百科）。

- 需要了解哈希的概念。

如果对哈希不了解，可以查看知乎的内容，网址如下：

<https://www.zhihu.com/question/26762707>

- 需要了解HTTP的工作原理。

不懂的话，自行百度或谷歌。

- 安装支持HTTP协议的客户端

如Postman或者cURL，其他也可以。

- 需要安装Python3.6+等文件

后面的章节"5、编译Python文件"会详细讲解。

- 源码地址

```
https://github.com/dvf/blockchain
```

2、创建一个区块链

可以用各种文本编辑器或者IDE编写Python程序，比如Subline、EditPlus、PyCharm、VSCode、Atom等。

新建一个目录，暂且命名为"py_blockchain_demo"，用于存放要编写的python文件，然后新建一个名为'blockchain.py'的文件。

2.1 描述区块链

创建一个 `Blockchain` 类，构造函数创建了一个初始化的空列表（要存储我们的区块链），并且另一个存储交易。下面是我们这个类的实例：

```
class Blockchain(object):
    def __init__(self):
        self.chain = []
        self.current_transactions = []

    def new_block(self):
        # Creates a new Block and adds it to the chain
        pass

    def new_transaction(self):
        # Adds a new transaction to the list of transactions
        pass

    @staticmethod
    def hash(block):
        # Hashes a Block
        pass

    @property
    def last_block(self):
        # Returns the last Block in the chain
        pass
```

`Blockchain` 类负责管理链式数据，它会存储交易并且还有添加新的区块到链式数据的 Method。让我们开始扩充更多方法。

2.2 区块结构

每个块都有一个 索引，一个 时间戳（Unix时间戳），一个 事务列表，一个 校验（稍后详述）和前一个块的散列。

下面是一个Block的例子：

```
block = {
    'index': 1,
    'timestamp': 1506057125.900785,
    'transactions': [
        {
            'sender': "8527147fe1f5426f9dd545de4b27ee00",
            'recipient': "a77f5cdfa2934df3954a5c7c7da5df1f",
            'amount': 5,
        }
    ]
}
```

```
],
  'proof': 324984774000,
  'previous_hash': "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e730433
62938b9824"
}
```

在这一点上，一个 **区块链** 的概念应该是明显的 - 每个新块都包含在其内的前一个块的 **散列**。这是至关重要的，因为这是 **区块链** 不可改变的原因：如果攻击者损坏 **区块链** 中较早的块，则所有后续块将包含不正确的哈希值。

这有道理吗？如果你还没有想通，花点时间仔细思考一下 - 这是区块链背后的核心理念

2.3 添加交易到区块

我们将需要一个添加交易到区块的方式。我们的 `new_transaction()` 方法的责任就是这个，并且它非常的简单：

```
class Blockchain(object):
    ...

    def new_transaction(self, sender, recipient, amount):
        """
        Creates a new transaction to go into the next mined Block
        :param sender: <str> Address of the Sender
        :param recipient: <str> Address of the Recipient
        :param amount: <int> Amount
        :return: <int> The index of the Block that will hold this transactio
n
        """

        self.current_transactions.append({
            'sender': sender,
            'recipient': recipient,
            'amount': amount,
        })

        return self.last_block['index'] + 1
```

`new_transaction()` 方法添加了交易到列表，它返回了交易将被添加到的区块的索引---讲开采下一个这对稍后对提交交易的用户有用。

2.4 创建新的区块

当我们的 **Blockchain** 被实例化后，我们需要将 **创世** 区块（一个没有前导区块的区块）添加进去。我们还需要向我们的起源块添加一个 **证明**，这是挖矿的结果(或工作证明)。我们稍后会

详细讨论挖矿。

除了在构造函数中创建 创世 区块外，我们还会补全 `new_block()`、`new_transaction()` 和 `hash()` 函数：

```
import hashlib
import json
from time import time

class Blockchain(object):
    def __init__(self):
        self.current_transactions = []
        self.chain = []

        # 创建创世区块
        self.new_block(previous_hash=1, proof=100)

    def new_block(self, proof, previous_hash=None):
        """
        创建一个新的区块到区块链中
        :param proof: <int> 由工作证明算法生成的证明
        :param previous_hash: (Optional) <str> 前一个区块的 hash 值
        :return: <dict> 新区块
        """

        block = {
            'index': len(self.chain) + 1,
            'timestamp': time(),
            'transactions': self.current_transactions,
            'proof': proof,
            'previous_hash': previous_hash or self.hash(self.chain[-1]),
        }

        # 重置当前交易记录
        self.current_transactions = []

        self.chain.append(block)
        return block

    def new_transaction(self, sender, recipient, amount):
        """
        创建一笔新的交易到下一个被挖掘的区块中
        :param sender: <str> 发送人的地址
        :param recipient: <str> 接收人的地址
        :param amount: <int> 金额
        :return: <int> 持有本次交易的区块索引
        """
```

```

        self.current_transactions.append({
            'sender': sender,
            'recipient': recipient,
            'amount': amount,
        })

        return self.last_block['index'] + 1

@property
def last_block(self):
    return self.chain[-1]

@staticmethod
def hash(block):
    """
    给一个区块生成 SHA-256 值
    :param block: <dict> Block
    :return: <str>
    """

    # 我们必须确保这个字典（区块）是经过排序的，否则我们将会得到不一致的散列
    block_string = json.dumps(block, sort_keys=True).encode()
    return hashlib.sha256(block_string).hexdigest()

```

上面的代码应该是直白的 --- 为了让代码清晰，我添加了一些注释和文档说明。我们差不多完成了我们的区块链。但在这个时候你一定很疑惑新的块是怎么被创建、锻造或挖掘的。

3、工作量证明算法

3.1 理解工作量证明

使用工作量证明（PoW）算法，来证明是如何在区块链上创建或挖掘新的区块。PoW 的目标是计算出一个符合特定条件的数字，这个数字对于所有人而言必须在计算上非常困难，但易于验证。这是工作证明背后的核心思想。

我们将看到一个简单的例子帮助你理解：

假设一个整数 x 乘以另一个整数 y 的积的 Hash 值必须以 0 结尾，即 $\text{hash}(x * y) = \text{ac23dc}...0$ 。设 $x = 5$ ，求 y ？用 Python 实现：

```

from hashlib import sha256
x = 5
y = 0 # We don't know what y should be yet...
while sha256(f'{x*y}'.encode()).hexdigest()[-1] != "0":
    y += 1

```

```
print(f'The solution is y = {y}')
```

结果是：y = 21。因为，生成的 Hash 值结尾必须为 0。

```
hash(5 * 21) = 1253e9373e...5e3600155e860
```

在比特币中，工作量证明算法被称为 [Hashcash](#)，它和上面的问题很相似，只不过计算难度非常大。这就是矿工们为了争夺创建区块的权利而争相计算的问题。通常，计算难度与目标字符串需要满足的特定字符的数量成正比，矿工算出结果后，就会获得一定数量的比特币奖励（通过交易）。

验证结果，当然非常容易。

3.2 实现工作量证明

让我们来实现一个相似 PoW 算法。规则类似上面的例子：

找到一个数字 P，使得它与前一个区块的 proof 拼接成的字符串的 Hash 值以 4 个零开头。

```
import hashlib
import json

from time import time
from uuid import uuid4

class Blockchain(object):
    ...

    def proof_of_work(self, last_proof):
        """
        Simple Proof of Work Algorithm:
        - Find a number p' such that hash(pp') contains leading 4 zeroes, w
        here p is the previous p'
        - p is the previous proof, and p' is the new proof
        :param last_proof: <int>
        :return: <int>
        """

        proof = 0
        while self.valid_proof(last_proof, proof) is False:
            proof += 1

        return proof

    @staticmethod
```

```

def valid_proof(last_proof, proof):
    """
    Validates the Proof: Does hash(last_proof, proof) contain 4 leading
    zeroes?

    :param last_proof: <int> Previous Proof
    :param proof: <int> Current Proof
    :return: <bool> True if correct, False if not.
    """

    guess = f'{last_proof}{proof}'.encode()
    guess_hash = hashlib.sha256(guess).hexdigest()
    return guess_hash[:4] == "0000"

```

衡量算法复杂度的办法是修改零开头的个数。使用 4 个来用于演示，你会发现多一个零都会大大增加计算出结果所需的时间。

现在 Blockchain 类基本已经完成了，接下来使用 HTTP requests 来进行交互。

4、Blockchain 作为 API 接口

我们将使用 Python Flask 框架，这是一个轻量 Web 应用框架，它方便将网络请求映射到 Python 函数，现在我们来让 Blockchain 运行在基于 Flask web 上。

我们将创建三个接口：

- /transactions/new：创建一个交易并添加到区块
- /mine：告诉服务器去挖掘新的区块
- /chain：返回整个区块链

4.1 创建节点

我们的“Flask 服务器”将扮演区块链网络中的一个节点。我们先添加一些框架代码：

```

import hashlib
import json
from textwrap import dedent
from time import time
from uuid import uuid4

from flask import Flask

class Blockchain(object):
    ...

# 实例化节点

```

```

app = Flask(__name__)

# 为此节点生成一个全球唯一的地址
node_identifier = str(uuid4()).replace('-', '')

# 实例化 Blockchain 类
blockchain = Blockchain()

# 创建 /mine 接口, GET 方式请求
@app.route('/mine', methods=['GET'])
def mine():
    return "We'll mine a new Block"

# 创建 /transactions/new 接口, POST 方式请求, 可以给接口发送交易数据
@app.route('/transactions/new', methods=['POST'])
def new_transaction():
    return "We'll add a new transaction"

# 创建 /chain 接口, 返回整个区块链
@app.route('/chain', methods=['GET'])
def full_chain():
    response = {
        'chain': blockchain.chain,
        'length': len(blockchain.chain),
    }
    return jsonify(response), 200

# 服务器运行端口 5000
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)

```

4.2 发送交易

发送到节点的交易数据结构如下：

```

{
    "sender": "my address",
    "recipient": "someone else's address",
    "amount": 5
}

```

因为我们已经有了添加交易的方法，所以基于接口来添加交易就很简单了。让我们为添加事务写函数：

```

import hashlib

```



```

import json
from textwrap import dedent
from time import time
from uuid import uuid4

from flask import Flask, jsonify, request

...

@app.route('/transactions/new', methods=['POST'])
def new_transaction():
    values = request.get_json()

    # Check that the required fields are in the POST'ed data
    required = ['sender', 'recipient', 'amount']
    if not all(k in values for k in required):
        return 'Missing values', 400

    # Create a new Transaction
    index = blockchain.new_transaction(values['sender'], values['recipient'],
    , values['amount'])

    response = {'message': f'Transaction will be added to Block {index}'}
    return jsonify(response), 201

```

4.3 挖矿

挖矿正是神奇所在，它很简单，做了一下三件事：

1. 计算工作量证明 PoW
2. 通过新增一个交易授予矿工（自己）一个币
3. 构造新区块并将其添加到链中

```

import hashlib
import json

from time import time
from uuid import uuid4

from flask import Flask, jsonify, request

...

@app.route('/mine', methods=['GET'])
def mine():
    # We run the proof of work algorithm to get the next proof...

```

```

last_block = blockchain.last_block
last_proof = last_block['proof']
proof = blockchain.proof_of_work(last_proof)

# We must receive a reward for finding the proof.
# The sender is "0" to signify that this node has mined a new coin.
blockchain.new_transaction(
    sender="0",
    recipient=node_identifier,
    amount=1,
)

# Forge the new Block by adding it to the chain
previous_hash = blockchain.hash(last_block)
block = blockchain.new_block(proof, previous_hash)

response = {
    'message': "New Block Forged",
    'index': block['index'],
    'transactions': block['transactions'],
    'proof': block['proof'],
    'previous_hash': block['previous_hash'],
}
return jsonify(response), 200

```

注意交易的接收者是我们自己的服务器节点，我们做的大部分工作都只是围绕 Blockchain 类方法进行交互。到此，我们的区块链就算完成了，我们来实际运行下。

5、编译Python文件

在Mac上编译Python3这里有点麻烦，特别说明一下。

5.1 Mac安装Python3

Python之所以强大，其中一个原因是其丰富的第三方库。pip则是python第三方库的包管理工具。

由于在Mac上python2和python3是共存的。因而python3对应的包管理工具的命令就是pip3。

如果通过Homebrew安装python3，那么pip3会同时安装。所以建议直接通过homebrew安装python3。

```

# 安装Python3
brew install python3
# 检查Python3版本 (即检查是否安装成功)
python3 -V

```

5.2 安装Pipenv

Python版本众多，在开发中经常需要使用不同版本的Python，不同项目也可能会使用不同版本的第三方库，这是我们就需要搭建多个Python虚拟环境。pipenv为官方推荐工具，使Python虚拟环境更加方便。

```
# 安装pipenv
pip3 install pipenv
```

5.3 创建Python虚拟环境

进入到Python文件所在目录，依次执行如下命令

```
# 创建Python虚拟环境，并指定Python版本
pipenv --python=python3.6
# 安装Flask、requests模块
pipenv install Flask==0.12.2 requests==2.18.4
```

6、运行区块链

可以使用 cURL 或 Postman 去和 API 进行交互。

用pipenv指定端口号（代码中默认端口为5000，可以省略后面的"-p 5000"），并启动Sever：

```
pipenv run python blockchain.py -p 5000
```

终端显示如下：

```
wenzildeiMac:py_blockchain_demo wenzil$ pipenv run python blockchain.py -p 5000
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

让我们通过请求 <http://localhost:5000/mine> （GET）来进行挖矿：

The screenshot shows the Postman interface with a GET request to `http://localhost:5000/mine` sent successfully. The response is a JSON object with the following structure:

```
{
  "index": 2,
  "message": "New Block Forged",
  "previous_hash": "6218bf785fbbdd0365118f06c1f9c52dc2eadd29e935678aa23bd28257046882",
  "proof": 466,
  "transactions": [
    {
      "amount": 1,
      "recipient": "d9878b6ee94c470682e7a7c7467dc450",
      "sender": "0"
    }
  ]
}
```

The status is 200 OK, with a response time of 27 ms and a size of 437 B. The response is displayed in the Pretty JSON format.

用 Postman 发起一个 GET 请求。

创建一个交易请求，请求 <http://localhost:5000/transactions/new>（POST），如图：

交易请求 Examples (0) ▾

POST ▾

http://localhost:5000/transactions/new

Params

Send ▾

Save ▾

Authorization Headers (1) **Body** Pre-request Script Tests Cookies Code

☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary JSON (application/json) ▾

```
1 {
2   "sender": "d4ee26eee15148ee92c6cd394edd974e",
3   "recipient": "someone-other-address",
4   "amount": 5
5 }
```



Body

Cookies

Headers (4)

Test Results

Status: 201 CREATED Time: 24 ms Size: 207 B

Pretty Raw Preview JSON ▾   Save Response

```
1 {
2   "message": "Transaction will be added to Block 3"
3 }
```

"Headers"中添加这样的Key-Value，返回会报服务器错误"500 Internal Server Error"。

```
'content-type': 'application/json'
```

如果不是使用 Postman，则用一下的 cURL 语句也是一样的：

```
$ curl -X POST -H "Content-Type: application/json" -d '{
  "sender": "d4ee26eee15148ee92c6cd394edd974e",
  "recipient": "someone-other-address",
  "amount": 5
}' "http://localhost:5000/transactions/new"
```

提交了两次请求后，需要挖矿（也就是请求"mine"接口）使交易生效。

GET
http://localhost:5000/mine
Params
Send
Save

Authorization
Headers
Body
Pre-request Script
Tests
Cookies
Code

Key	Value	Description	...	Bulk Edit	Presets
New key	Value	Description			

Body
Cookies
Headers (4)
Test Results
Status: 200 OK
Time: 47 ms
Size: 698 B

Pretty
Raw
Preview
JSON
Save Response

```

1 {
2   "index": 3,
3   "message": "New Block Forged",
4   "previous_hash": "0cfdac7a19302f5d3af7d20ebf7997afadd88b5f4f937ade867c24701fc21158",
5   "proof": 9425,
6   "transactions": [
7     {
8       "amount": 5,
9       "recipient": "someone-other-address",
10      "sender": "d4ee26eee15148ee92c6cd394edd974e"
11    },
12    {
13      "amount": 5,
14      "recipient": "someone-other-address",
15      "sender": "d4ee26eee15148ee92c6cd394edd974e"
16    },
17    {
18      "amount": 1,
19      "recipient": "2aaa370d7d604b708678a510f11da5d7",
20      "sender": "0"
21    }
22  ]
23 }

```

这时，再通过请求 <http://localhost:5000/chain> 可以得到所有的块信息。

```

{
  "chain": [
    {
      "index": 1,
      "previous_hash": "1",
      "proof": 100,
      "timestamp": 1528705939.274652,
      "transactions": []
    },
    {
      "index": 2,
      "previous_hash": "22efcab1c2990f7d371e5159fe9f753ea4f5b0f9b173f0e41004eb1b6efcd1f7",
      "proof": 24960,
      "timestamp": 1528705943.739656,
      "transactions": [
        {
          "amount": 1,

```

```

        "recipient": "2aaa370d7d604b708678a510f11da5d7",
        "sender": "0"
    }
]
},
{
    "index": 3,
    "previous_hash": "0cfdac7a19302f5d3af7d20ebf7997afadd88b5f4f937a
de867c24701fc21158",
    "proof": 9425,
    "timestamp": 1528706168.36694,
    "transactions": [
        {
            "amount": 5,
            "recipient": "someone-other-address",
            "sender": "d4ee26eee15148ee92c6cd394edd974e"
        },
        {
            "amount": 5,
            "recipient": "someone-other-address",
            "sender": "d4ee26eee15148ee92c6cd394edd974e"
        },
        {
            "amount": 1,
            "recipient": "2aaa370d7d604b708678a510f11da5d7",
            "sender": "0"
        }
    ]
}
],
"length": 3
}

```

7、一致性（共识）

我们已经有了一个基本的区块链可以接受交易和挖矿。但是区块链系统应该是分布式的。既然是分布式的，那么我们究竟拿什么保证所有节点有同样的链呢？这就是一致性问题，我们要想在网络上有多个节点，就必须实现一个一致性的算法

7.1 注册节点

在实现一致性算法之前，我们需要找到一种方式让一个节点知道它相邻的节点。每个节点都需要保存一份包含网络中其它节点的记录。因此让我们新增几个接口：

- 1、/nodes/register：接收 URL 形式的新节点列表。
- 2、/nodes/resolve：执行一致性算法，解决任何冲突，确保节点拥有正确的链。

我们修改下 Blockchain 的 init 函数并提供一个注册节点方法：

```
...
from urllib.parse import urlparse
...

class Blockchain(object):
    def __init__(self):
        ...
        self.nodes = set()
        ...

    def register_node(self, address):
        """
        Add a new node to the list of nodes
        :param address: <str> Address of node. Eg. 'http://192.168.0.5:5000'
        :return: None
        """

        parsed_url = urlparse(address)
        self.nodes.add(parsed_url.netloc)
```

我们用 set 来储存节点，这是一种避免重复添加节点的简单方法。

7.2 实现共识算法

就像先前讲的那样，当一个节点与另一个节点有不同的链时，就会产生冲突。为了解决这个问题，我们将制定最长的有效链条是最权威的规则。换句话说就是：在这个网络里最长的链就是最权威的。我们将使用这个算法，在网络中的节点之间达成共识。

```
...
import requests

class Blockchain(object):
    ...

    def valid_chain(self, chain):
        """
        Determine if a given blockchain is valid
        :param chain: <list> A blockchain
        :return: <bool> True if valid, False if not
        """

        last_block = chain[0]
        current_index = 1
```



```

while current_index < len(chain):
    block = chain[current_index]
    print(f'{last_block}')
    print(f'{block}')
    print("\n-----\n")
    # Check that the hash of the block is correct
    if block['previous_hash'] != self.hash(last_block):
        return False

    # Check that the Proof of Work is correct
    if not self.valid_proof(last_block['proof'], block['proof']):
        return False

    last_block = block
    current_index += 1

```

```

return True

```

```

def resolve_conflicts(self):

```

```

    """

```

```

    This is our Consensus Algorithm, it resolves conflicts
    by replacing our chain with the longest one in the network.
    :return: <bool> True if our chain was replaced, False if not
    """

```

```

    neighbours = self.nodes
    new_chain = None

```

```

    # We're only looking for chains longer than ours
    max_length = len(self.chain)

```

```

    # Grab and verify the chains from all the nodes in our network

```

```

    for node in neighbours:
        response = requests.get(f'http://{node}/chain')

```

```

        if response.status_code == 200:
            length = response.json()['length']
            chain = response.json()['chain']

```

```

            # Check if the length is longer and the chain is valid

```

```

            if length > max_length and self.valid_chain(chain):
                max_length = length
                new_chain = chain

```

```

    # Replace our chain if we discovered a new, valid chain longer than

```

ours

```

    if new_chain:
        self.chain = new_chain

```

```
return True
```

```
return False
```

第一个方法 `valid_chain()` 负责检查一个链是否有效，方法是遍历每个块并验证散列和证明。

`resolve_conflicts()` 是一个遍历我们所有邻居节点的方法，下载它们的链并使用上面的方法验证它们。如果找到一个长度大于我们的有效链条，我们就取代我们的链条。

我们将两个端点注册到我们的API中，一个用于添加相邻节点，另一个用于解决冲突：

```
@app.route('/nodes/register', methods=['POST'])
def register_nodes():
    values = request.get_json()

    nodes = values.get('nodes')
    if nodes is None:
        return "Error: Please supply a valid list of nodes", 400

    for node in nodes:
        blockchain.register_node(node)

    response = {
        'message': 'New nodes have been added',
        'total_nodes': list(blockchain.nodes),
    }
    return jsonify(response), 201

@app.route('/nodes/resolve', methods=['GET'])
def consensus():
    replaced = blockchain.resolve_conflicts()

    if replaced:
        response = {
            'message': 'Our chain was replaced',
            'new_chain': blockchain.chain
        }
    else:
        response = {
            'message': 'Our chain is authoritative',
            'chain': blockchain.chain
        }

    return jsonify(response), 200
```

在这一点上，如果你喜欢，你可以使用一台不同的机器，并在你的网络上启动不同的节点。或者

使用同一台机器上的不同端口启动进程。我在我的机器上，不同的端口上创建了另一个节点，并将其注册到当前节点。因此，我有两个节点：`http://localhost:5000` 和 `http://localhost:5001`。

注册一个新节点

Examples (0)

POST

http://localhost:5000/nodes/register

Params

Send

Save

Authorization

Headers (1)

Body

Pre-request Script

Tests

Cookies

Code

	Key	Value	Description	...	Bulk Edit	Presets
<input checked="" type="checkbox"/>	content-type	application/json				
	New key	Value	Description			

Body

Cookies

Headers (4)

Test Results

Status: 201 CREATEDTime: 5 msSize: 242 B

Pretty

Raw

Preview

JSON

Save Response

1

{

2

"message": "New nodes have been added",

3

"total_nodes": [

4

"127.0.0.1:5001"

5

]

6

}

然后我在节点 2 上挖掘了一些新的块，以确保链条更长。之后，我在节点1上调用 `GET /nodes/resolve`，其中区块链数据由一致性算法取代（注：这一步运行结果跟原文有点出入，可以查看原文的结果）：

GET http://localhost:5000/nodes/resolve Params Send

Authorization Headers (1) Body Pre-request Script Tests

Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/> content-type	application/json			
New key	Value	Description		

Body Cookies Headers (4) Test Results Status: 200 OK Time: 25 ms

Pretty Raw Preview JSON

```

1 {
2   "chain": [
3     {
4       "index": 1,
5       "previous_hash": "1",
6       "proof": 100,
7       "timestamp": 1528711563.4878132,
8       "transactions": []
9     }
10  ],
11  "message": "Our chain is authoritative"
12 }

```

这是一个包，去找一些朋友一起，以帮助测试你的区块链。

最新完整源代码如下：

```

import hashlib
import json
from time import time
from urllib.parse import urlparse
from uuid import uuid4

import requests
from flask import Flask, jsonify, request

class Blockchain:
    def __init__(self):
        self.current_transactions = []
        self.chain = []
        self.nodes = set()

        # Create the genesis block
        self.new_block(previous_hash='1', proof=100)

    def register_node(self, address):
        """

```

```
Add a new node to the list of nodes
:param address: Address of node. Eg. 'http://192.168.0.5:5000'
"""
```

```
parsed_url = urlparse(address)
if parsed_url.netloc:
    self.nodes.add(parsed_url.netloc)
elif parsed_url.path:
    # Accepts an URL without scheme like '192.168.0.5:5000'.
    self.nodes.add(parsed_url.path)
else:
    raise ValueError('Invalid URL')
```

```
def valid_chain(self, chain):
    """
```

```
Determine if a given blockchain is valid
:param chain: A blockchain
:return: True if valid, False if not
    """
```

```
last_block = chain[0]
current_index = 1
```

```
while current_index < len(chain):
    block = chain[current_index]
    print(f'{last_block}')
    print(f'{block}')
    print("\n-----\n")
    # Check that the hash of the block is correct
    last_block_hash = self.hash(last_block)
    if block['previous_hash'] != last_block_hash:
        return False
```

```
    # Check that the Proof of Work is correct
    if not self.valid_proof(last_block['proof'], block['proof'], last_block_hash):
        return False
```

```
    last_block = block
    current_index += 1
```

```
return True
```

```
def resolve_conflicts(self):
    """
```

```
This is our consensus algorithm, it resolves conflicts
by replacing our chain with the longest one in the network.
```

```

: return: True if our chain was replaced, False if not
"""

neighbours = self.nodes
new_chain = None

# We're only looking for chains longer than ours
max_length = len(self.chain)

# Grab and verify the chains from all the nodes in our network
for node in neighbours:
    response = requests.get(f'http://{node}/chain')

    if response.status_code == 200:
        length = response.json()['length']
        chain = response.json()['chain']

        # Check if the length is longer and the chain is valid
        if length > max_length and self.valid_chain(chain):
            max_length = length
            new_chain = chain

# Replace our chain if we discovered a new, valid chain longer than
ours
if new_chain:
    self.chain = new_chain
    return True

return False

def new_block(self, proof, previous_hash):
    """
    Create a new Block in the Blockchain
    :param proof: The proof given by the Proof of Work algorithm
    :param previous_hash: Hash of previous Block
    :return: New Block
    """

    block = {
        'index': len(self.chain) + 1,
        'timestamp': time(),
        'transactions': self.current_transactions,
        'proof': proof,
        'previous_hash': previous_hash or self.hash(self.chain[-1]),
    }

    # Reset the current list of transactions
    self.current_transactions = []

```

```
self.chain.append(block)
return block
```

```
def new_transaction(self, sender, recipient, amount):
    """
```

```
Creates a new transaction to go into the next mined Block
:param sender: Address of the Sender
:param recipient: Address of the Recipient
:param amount: Amount
:return: The index of the Block that will hold this transaction
    """
```

```
self.current_transactions.append({
    'sender': sender,
    'recipient': recipient,
    'amount': amount,
})
```

```
return self.last_block['index'] + 1
```

```
@property
```

```
def last_block(self):
    return self.chain[-1]
```

```
@staticmethod
```

```
def hash(block):
    """
```

```
Creates a SHA-256 hash of a Block
:param block: Block
    """
```

```
# We must make sure that the Dictionary is Ordered, or we'll have in
consistent hashes
```

```
block_string = json.dumps(block, sort_keys=True).encode()
return hashlib.sha256(block_string).hexdigest()
```

```
def proof_of_work(self, last_block):
    """
```

```
Simple Proof of Work Algorithm:
- Find a number p' such that hash(pp') contains leading 4 zeroes
- Where p is the previous proof, and p' is the new proof
```

```
:param last_block: <dict> last Block
:return: <int>
    """
```

```
last_proof = last_block['proof']
last_hash = self.hash(last_block)
```

```
proof = 0
while self.valid_proof(last_proof, proof, last_hash) is False:
    proof += 1
```

```
return proof
```

```
@staticmethod
```

```
def valid_proof(last_proof, proof, last_hash):
    """
```

```
    Validates the Proof
```

```
    :param last_proof: <int> Previous Proof
```

```
    :param proof: <int> Current Proof
```

```
    :param last_hash: <str> The hash of the Previous Block
```

```
    :return: <bool> True if correct, False if not.
```

```
    """
```

```
    guess = f'{last_proof}{proof}{last_hash}'.encode()
```

```
    guess_hash = hashlib.sha256(guess).hexdigest()
```

```
    return guess_hash[:4] == "0000"
```

```
# Instantiate the Node
```

```
app = Flask(__name__)
```

```
# Generate a globally unique address for this node
```

```
node_identifier = str(uuid4()).replace('-', '')
```

```
# Instantiate the Blockchain
```

```
blockchain = Blockchain()
```

```
@app.route('/mine', methods=['GET'])
```

```
def mine():
```

```
    # We run the proof of work algorithm to get the next proof...
```

```
    last_block = blockchain.last_block
```

```
    proof = blockchain.proof_of_work(last_block)
```

```
    # We must receive a reward for finding the proof.
```

```
    # The sender is "0" to signify that this node has mined a new coin.
```

```
    blockchain.new_transaction(
```

```
        sender="0",
```

```
        recipient=node_identifier,
```

```
        amount=1,
```

```
)
```

```
    # Forge the new Block by adding it to the chain
```

```
    previous_hash = blockchain.hash(last_block)
```



```

block = blockchain.new_block(proof, previous_hash)

response = {
    'message': "New Block Forged",
    'index': block['index'],
    'transactions': block['transactions'],
    'proof': block['proof'],
    'previous_hash': block['previous_hash'],
}
return jsonify(response), 200

@app.route('/transactions/new', methods=['POST'])
def new_transaction():
    values = request.get_json()

    # Check that the required fields are in the POST'ed data
    required = ['sender', 'recipient', 'amount']
    if not all(k in values for k in required):
        return 'Missing values', 400

    # Create a new Transaction
    index = blockchain.new_transaction(values['sender'], values['recipient'],
    values['amount'])

    response = {'message': f'Transaction will be added to Block {index}'}
    return jsonify(response), 201

@app.route('/chain', methods=['GET'])
def full_chain():
    response = {
        'chain': blockchain.chain,
        'length': len(blockchain.chain),
    }
    return jsonify(response), 200

@app.route('/nodes/register', methods=['POST'])
def register_nodes():
    values = request.get_json()

    nodes = values.get('nodes')
    if nodes is None:
        return "Error: Please supply a valid list of nodes", 400

    for node in nodes:
        blockchain.register_node(node)

```

```

response = {
    'message': 'New nodes have been added',
    'total_nodes': list(blockchain.nodes),
}
return jsonify(response), 201

@app.route('/nodes/resolve', methods=['GET'])
def consensus():
    replaced = blockchain.resolve_conflicts()

    if replaced:
        response = {
            'message': 'Our chain was replaced',
            'new_chain': blockchain.chain
        }
    else:
        response = {
            'message': 'Our chain is authoritative',
            'chain': blockchain.chain
        }

    return jsonify(response), 200

if __name__ == '__main__':
    from argparse import ArgumentParser

    parser = ArgumentParser()
    parser.add_argument('-p', '--port', default=5000, type=int, help='port to listen on')
    args = parser.parse_args()
    port = args.port

    app.run(host='0.0.0.0', port=port)

```

本文内容大部分来自如下翻译文章，重新运行截图，作了部分修改，如有侵权联系我删除文章：
[使用 Python 一步步搭建自己的区块链](#)