מבינים BlockChain דרך הידיים

מאת הודיה ק.

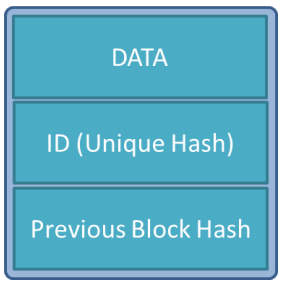
הקדמה

Blockchain כשמו כן הוא - שרשרת בלוקים המכילים מידע. זו טכנולוגיה שהוכרזה בשנת 1991 ע"י קבוצת חוקרים, אך למעשה אנו מכירים אותה רק משנת 2009, כאשר השתמשו בה במטבע הדיגיטלי bitcoin. במאמר זה אסביר את הרעיון של טכנולוגיית ה-blockchain וכן אממש את העקרונות הבסיסיים בשפת פייתון.

הרעיון של ה-blockchain הוא לאפשר אמינות נתונים, ע"י שמירתם באופן מפוזר בצמתי הרשת במקומות שונים. אם מישהו ינסה לשנות בלוק במופע אחד של מסד הנתונים, הצמתים האחרים לא ישתנו וכך ימנעו מלקבל את השינוי בבלוק, ואמינות הנתונים לא תפגע. אם משתמש אחד מתעסק ברישום העסקאות של bitcoin - משנה את ה-DATA של המטבעות למשל, כל הצמתים האחרים יצלבו את המידע שלהם זה בזה ויזהו בקלות את הבלוק עם המידע השגוי.

מערכת זו מסייעת לקבוע סדר מדויק ושקוף של אירועים. בדרך זו, אף צומת אחד בתוך הרשת לא יכול לשנות מידע המוחזק בתוכה. מכיוון שכך, הנתונים המוחזקים ב-blockchain הם "בלתי הפיכים".

אז, איך זה בנוי?



כל בלוק מכיל:

1. מידע - DATA
2. מזהה ייחודי - המחושב ע"י פונקציית hash על נתוני הבלוק.
3. Hash של הבלוק שקדם לו - מה שיוצר את שרשרת הבלוקים.

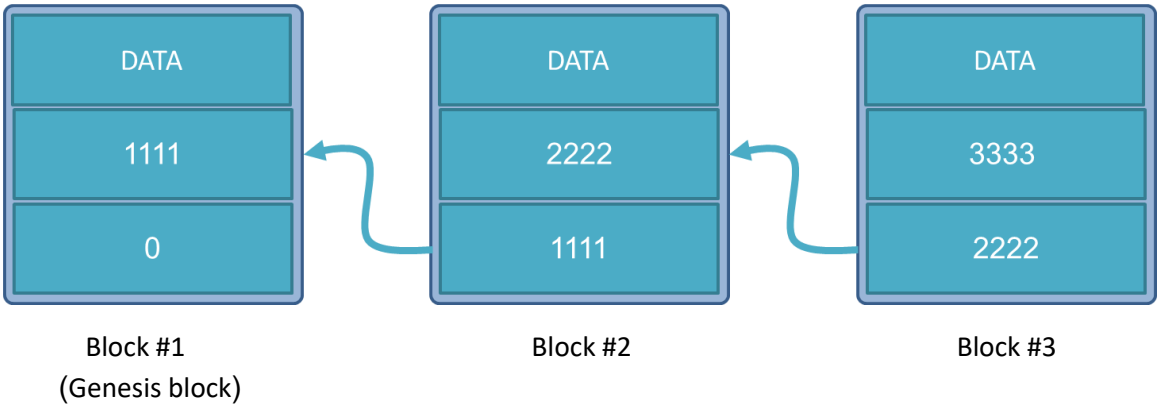
המידע הנמצא בכל בלוק משתנה בהתאם לסוג של ה-blockchain, לדוגמא עבור bitcoin, ה-data יכיל למשל את כמות המטבעות, מי השולח ומי הנמען.

המזהה הייחודי, הוא בעצם כמו id של הבלוק, והוא מחושב ע"י פונקציית hash. לפונקציה שולחים את כל מה שהבלוק מכיל - כלומר את ה-data ואת ה-hash של הבלוק הקודם, והפונקציה מחשבת לפי הנתונים את ה-hash הייחודי לבלוק.

מיד כאשר נוצר בלוק, המזהה שלו מחושב ע"י פונקציית ה-hash, וכן כאשר מידע השתנה בתוך הבלוק ה-hash ישתנה גם הוא. כלומר ה-hash מאד שימושי כאשר נרצה לזהות שינויים בבלוק.

הבלוק הראשון נקרא Genesis block, ומה שייחודיי בו זה שה-Previous block hash יהיה שווה לאפס, כי כאמור אין לפניו שום בלוק.

ננסה להראות מה הסברנו עד כה:



[כמובן שמספרי ה-hash יהיו הרבה יותר ארוכים ומסובכים, אך בשביל הדוגמא הפשטתי את העניין]

ניתן לראות שכל בלוק מכיל את ה-hash של קודמו - מה שיוצר לנו את שרשרת הבלוקים, והבלוק הראשון מכיל אפס.

כעת, נדמיין שאקר ניסה לחבל בבלוק מספר 2, ושינה את מספר המטבעות ב-data. כתוצאה מכך כאמור ה-hash של הבלוק יחושב שוב (שהרי הוא מורכב מנתוני הבלוק ואם אחד שונה אזי ה-hash ישתנה). מה שיקרה זה שכבר בלוק 3 מצביע ל-previous block שלא קיים, וכך בעצם כל השרשרת 'תשבר' בנקודה זו - מבלוק 2 והלאה. כל שאר הבלוקים יהפכו להיות invalid מכיוון שהם לא מכילים מספר hash תקין של previous block.

כלומר, שינוי בלוק יחיד יגרום לכל הבלוקים העוקבים להיות invalid.

נחשוב רגע: מה הבעיה לקחת מחשב בעל כוח עיבוד חזק ולחשב את כל ה-hash של הבלוקים הבאים מחדש במהירות כך שה-blockchain תהיה שוב תקינה? אכן, זו אפשרות, ולכן ישנו מנגנון הגנה נוסף הנקרא **Proof of Work**.

Proof of Work (או בקיצור POW) זהו מנגנון המאט את יצירת הבלוקים. למשל ב-bitcoin זה לוקח כ-10 שניות על מנת לחשב מחדש את ה-POW הנדרש של בלוק ולהוסיף בלוק חדש לשרשרת.

זה מקשה על האקרים לחבל בבלוקים, מכיוון שאם הם מחבלים בבלוק כלשהו, הם יצטרכו לחשב מחדש את כל ה-POW של הבלוקים העוקבים, ודבר זה ייקח זמן. מגנון נוסף של אבטחה הוא שימוש ברשת P2P: כלומר peer to peer. כל אדם יכול להצטרך לרשת של הבלוק, וכאשר אדם מצטרף הוא מקבל 'העתק' של כל שרשרת הבלוקים אליו. וכך ישתמשו בזה לאמת שהבלוקים תקינים.

כאשר אדם יוצר בלוק חדש, הבלוק נשלח לכל אחד שנמצא ברשת, וכל אחד מאמת לפי מה שנמצא אצלו שאכן הבלוק תקין ולא חיבלו בו - ואם אכן זה כך הוא מוסיף את הבלוק לשרשרת בלוקים הנמצאת אצלו. דבר זה יוצר בדיקה שהבלוקים עקביים - ההסכמה אלו בלוקים תקינים ואלו לא. בלוקים שחיבלו בהם יודחו ע"י אנשים מהרשת.

אם כך, על מנת שהאקר יחבל ב-blockchain, הוא צריך לחבל בכל הבלוקים העוקבים בשרשרת, לחשב מחדש את ה-POW, ובנוסף לשלוט בלפחות 50 אחוז מהאנשים המשתתפים ברשת P2P. עדיין לא ברור? בואו נממש את עקרון ה-blockchain ב-python!

יאללה מימוש!

ראשית ניצור מחלקה לבלוק, כאשר כל בלוק יכיל את מיקומו בשרשרת, זמן היצירה שלו, מידע כלשהו, וכמובן את ה-hash של קודמו:

```
class Block:
    def __init__(self, index, time_span, data, previous_hash=''):
        self.index = index  # index in the chain
        self.time_span = time_span  # created time
        self.data = data  # some data
        self.previous_hash = previous_hash  # hash of the previous
            block in the chain

        self.hash = self.calculate_hash()  # current hash
```

נשים לב, בתחילה אותחל ה-hash של הבלוק הקודם במחרוזת ריקה, שכן רק כאשר יוצרים את שרשרת הבלוקים יודעים מהו ה-hash של הקודם, ולא כאשר יוצרים סתם בלוק בודד. על מנת לחשב את ה-hash של הבלוק, נשתמש בספרייה hashlib, הנועדה לספק אלגוריתמי גיבוב שונים. נכתוב את הפקודה:

```
import hashlib
```

השימוש בספרייה זו הוא כך: בוחרים מתוך הספרייה את פונקציית הגיבוב הרצויה (אנו נבחר sha256), מכניסים כפרמטר את המחרוזת מקודדת, ועל מנת להחזירה לאסקי - נבצע את הפעולה ההפוכה ע"י הפוקציה hexdigest בצורה הזו:

```
a_string = 'string to be calculate'
hashed_string = hashlib.sha256(a_string.encode()).hexdigest()
print(hashed_string)
```

והפלט יהיה תוצאת החישוב של פונקציה ה-hash על ה-string:

```
0ec9b482704b047fc42dbc318f6ea7f4009bb114f1ead0ac5f754e6cdfb10513
```

אם כך, נבצע שרשור פשוט של כל נתוני הבלוק וזה מה שנשלח לפונקציית hash:

```
def calculate_hash(self):
    all_block_data = str(self.index) + str(self.time_span) +
            self.data + self.previous_hash

    return hashlib.sha256(all_block_data.encode()).hexdigest()
```

לשם נוחות, נעמיס את str על מנת שנוכל להדפיס את הבלוקים:

```
def __str__(self):
    return 'block hash: ' + self.hash + '\n     at index: ' + \
           str(self.index) + ' created at: ' + self.time_span + \
           ' data: ' + self.data + '\n     previous hash: ' + \
           self.previous_hash
```

עד כה הקוד שכתבנו:

```
import hashlib

class Block:
    def __init__(self, index, time_span, data, previous_hash=''):
        self.index = index  # index in the chain
        self.time_span = time_span  # created time
        self.data = data  # some data
        self.previous_hash = previous_hash # hash of the previous block
                                           # in the chain
        self.hash = self.calculate_hash()  # current hash

    def calculate_hash(self):
        all_block_data = str(self.index) + str(self.time_span)
        + self.data + self.previous_hash
        return hashlib.sha256(all_block_data.encode()).hexdigest()

    def __str__(self):
        return 'block hash: ' + self.hash + '\n     at index: ' + \
               str(self.index) + ' created at: ' + self.time_span + \
               ' data: ' + self.data + '\n     previous hash: ' + \
               self.previous_hash
```

כעת נעבור ליצירת מחלקת blockchain - שתכיל רשימה של בלוקים:

```
class BlockChain:
    def __init__(self):
        self.chain = [self.create_genesis_block()]

def create_genesis_block(self):
        return Block(0, '01/01/2022', 'Genesis block', '0')
```

כאמור, הרשימה תאותחל בבלוק הראשון שנקרא Genesis block, ולשם כך יצרתי פונקציית עזר המחזירה לי את הבלוק הראשון, באינדקס אפס, כאשר הprevious hash שלו יהיה אפס. כעת, נוסיף פונקציה שמחזירה לי את הבלוק האחרון בשרשרת - על מנת שנוכל לגשת בקלות ל-hash של הבלוק האחרון:

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

ובנוסף, נוסיף פונקציה המאפשרת הוספת בלוק לשרשרת:

```
def add_block(self, new_block):
    new_block.previous_hash = self.get_last_block().hash
    # because we change the previous hash then the current hash need to
    # be calculate again
    new_block.hash = new_block.calculate_hash()
    self.chain.append(new_block)
```

נשים לב, כאשר מוסיפים בלוק חדש, צריך לערוך לו את ה-previous hash שיהיה ה-hash של הבלוק האחרון בשרשרת, ומכיוון שתוכן הבלוק השתנה - גם ה-hash של הבלוק צריך להיות מחושב מחדש. לבסוף נשרשר זאת לרשימת הבלוקים - chain.

כעת, כמו מקודם לשם נוחות נעמיס את הפונקציה str:

```
def __str__(self):
    chain = ''
    for block in self.chain:
        chain += str(block) + '\n'

    return chain
```

עד כה המחלקה של BlockChain תראה כך:

```
class BlockChain:
    def __init__(self):
        self.chain = [self.create_genesis_block()]

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

    def add_block(self, new_block):
        new_block.previous_hash = self.get_last_block().hash
        # because we change the previous hash then the current hash
                    need to be calculate again
        new_block.hash = new_block.calculate_hash()
        self.chain.append(new_block)

    def __str__(self):
        chain = ''
        for block in self.chain:
            chain += str(block) + '\n'
        return chain
    def create_genesis_block():
        return Block(0, '01/01/2022', 'Genesis block', '0')
```

לאחר שהגדרנו את שרשרת הבלוקים, נבדוק את הקוד שכתבנו. ניצור שרשרת בלוקים, נוסיף אליה מספר בלוקים ונראה את התוצאה:

```
bitcoin = BlockChain()
bitcoin.add_block(Block(1, '02/01/2022', 'amount = 5'))
bitcoin.add_block(Block(2, '03/01/2022', 'amount = 15'))
bitcoin.add_block(Block(3, '04/01/2022', 'amount = 25'))

print(bitcoin)
```

הפלט יהיה:

```
Block #0  block hash: 2a7c53c8f9c87b3f82f3d9afa56c4365b6b5558cde55daa6876c1dab016d26c5
               at index: 0 created at: 01/01/2022 data: Genesis block
               previous hash: 0
Block #1  block hash: c06c8c5a4b82fdd460b2cf2fb5577c6b6e816bf01426dc746524b9d28d4ac1ad
               at index: 1 created at: 02/01/2022 data: amount = 5
               previous hash: 2a7c53c8f9c87b3f82f3d9afa56c4365b6b5558cde55daa6876c1dab016d26c5
Block #2  block hash: a7c4432d0ae5f768d9537ce49fe95fe059a95d21ed64b5d4441f9f6bf7e45db7
               at index: 2 created at: 03/01/2022 data: amount = 15
               previous hash: c06c8c5a4b82fdd460b2cf2fb5577c6b6e816bf01426dc746524b9d28d4ac1ad
Block #3  block hash: 7cd1e5bae8dca37300b033557013dfd426358e939bf4667d46c1ef0761fb1274
               at index: 3 created at: 04/01/2022 data: amount = 25
               previous hash: a7c4432d0ae5f768d9537ce49fe95fe059a95d21ed64b5d4441f9f6bf7e45db7
```

ניתן לראות שכל בלוק מכיל את ה hash של ה-previous block.

יצרנו את הבסיס של ה-blockchain, כעת נרצה להוסיף פונקציה הבודקת האם ה-blockchain הוא תקין.

נעבור על כל הבלוקים בשרשרת, ונבדוק שתי בדיקות:

1. אם ה-hash שלה אכן זהה ל-hash שצריך להיות לה לפי הפונקציה calculate_hash().
2. שה-hash של ה-previous block אכן זהה לhash של הבלוק הקודם בשרשרת.

במידה ואחד מהבדיקות לא נכונה - נחזיר False. רק במקרה שעברנו על הכל והכל תקין - נחזיר True.

הפונקציה תראה כך:

```
def is_chain_valid(self):
    for i in range(1, len(self.chain)):
        current_block = self.chain[i]
        previous_block = self.chain[i-1]
        if current_block.hash != current_block.calculate_hash() or
current_block.previous_hash != previous_block.hash:
            return False

    return True
```

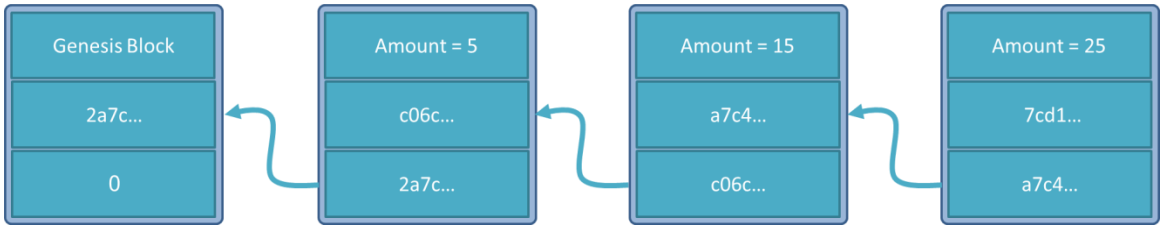
כעת נבדוק את השרשרת שיצרנו:

```
print('is chain valid? ' + str(bitcoin.is_chain_valid()))
```

הפלט:

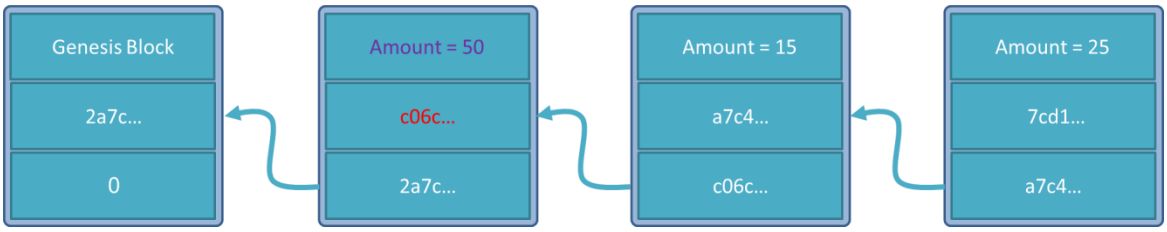
```
is chain valid? True
```

בוא ננסה לחבל בשרשרת, ולראות מה קורה. זה המצב ההתחלתי:



נשנה את המידע של הבלוק השני, במקום 5 מטבעות נשנה ל-50 מטבעות כך:

```
bitcoin.chain[1].data = '50'
```



כעת נבדוק את התקינות של השרשרת:

```
print('after tempering second block - is chain valid? ' +
str(bitcoin.is_chain_valid()))
```

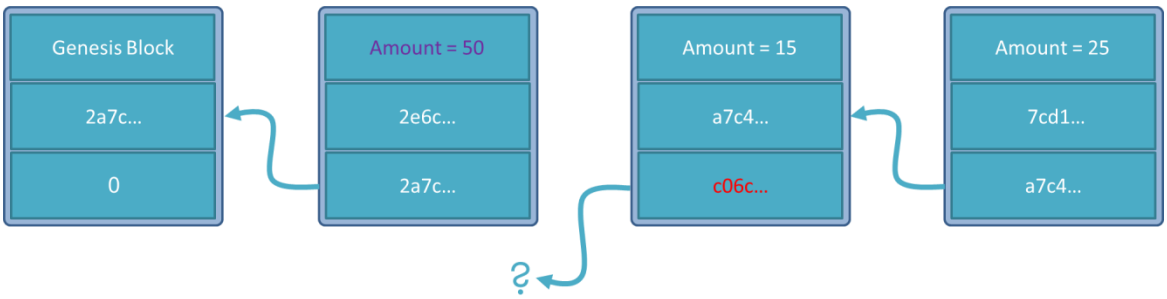
והפלט יהיה:

```
after tempering second block - is chain valid? False
```

הסיבה לכך היא שהבדיקה הראשונה לא תקינה - חישוב ה-hash לפי הפונקציה calculate_hash() **לא** זהה לערך ה-hash של הבלוק השני, מכיוון שתוכן הבלוק השתנה. כלומר, ה-hash של הבלוק (המסומן בצבע אדום) נשאר אותו הדבר גם לאחר שינוי ה-DATA.

אם כך, נוכל לחשוב - מה הבעיה, מלבד שינוי ה-data, נשנה גם את ה-hash של הבלוק הנוכחי- נחשבו מחדש כך:

```
bitcoin.chain[1].hash = bitcoin.chain[1].calculate_hash()
```



נריץ שוב את הבדיקה, ונגלה שעדיין הפלט הוא:

```
after tempering second block - is chain valid? False
```

הסיבה לכך כי הבדיקה השנייה לא תקינה - ה-Hash של הבלוק הבא בתור, בלוק מספר 3, מכיל בתוכו את ה-previous hash הלא מעודכן - לפני השינוי! כלומר, בלוק מספר 2 אכן תקין - כי חישבנו את ה-hash החדש שלו, אך כתוצאה מכך הוא ניתק את השרשרת.

כלומר ניתן לראות שאפשר להוסיף בלוק לשרשרת, אך לא לשנות או למחוק בלוק - כי אז השרשרת כבר לא תהיה תקינה. עדיין ישנה בעיית אבטחה: כי ניתן ע"י לולאה לחשב את כל הבלוקים העוקבים לבלוק שבו חיבלנו, לערוך להם את ה-previous hash ולחשב להם מחדש את ה-hash כך שהשרשרת תהיה תקינה.

על מנת להקשות על כך, כמו שהסברנו, נממש את מנגנון ה-Proof of Work!

Proof of Work

ב-bitcoin למשל, ה-POW מכריח שה-hash של הבלוק יתחיל עם מספר אפסים (ככל שיותר אפסים כך ייקח יותר זמן חישוב). נראה זאת:

```
def proof_of_work(self, difficulty):
    while self.hash[:difficulty] != ''.zfill(difficulty) :
        self.hash = self.calculate_hash()
```

בעצם ניצור לולאה, שמקבלת את ה-difficulty, כלומר את כמות האפסים הרצוי שיהיה בתחילת כל hash, וכל עוד תחילת ה-Hash שונה ממחרוזת האפסים באורך הרצוי (הפונקציה zfill ממלאת באפסים לפי הכמות שכותבים לה), אז נחשב מחדש את ה-hash.

על מנת שה-hash ישתנה וזו לא תהיה לולאה אינסופית, נצטרך להוסיף לבלוק עוד תכונה - נקרא לה nonce, ואותה נעלה ב-1 כל פעם, עד שנגיע לתוצאה הרצויה. אין לה שום משמעות, היא רק נועדה למימוש ה-POW. ראשית נוסיף זאת לבלוק, מאותחל באפס:

```
def __init__(self, index, time_span, data, previous_hash=''):
    self.index = index  # index in the chain
    self.time_span = time_span  # created time
    self.data = data  # some data
    self.previous_hash = previous_hash  # hash of the previous block in
the chain
    self.nonce = 0
    self.hash = self.calculate_hash()  # current hash
```

כעת נוסיף זאת לפונקציה של ה-POW:

```
def proof_of_work(self, difficulty):
    while self.hash[:difficulty] != ''.zfill(difficulty) :
        self.nonce += 1
        self.hash = self.calculate_hash()
```

וכמובן לא נשכח להוסיף זאת לחישוב של פונקציה ה-hash (אחרת דבר לא ישתנה):

```
def calculate_hash(self):
    all_block_data = str(self.index) + str(self.time_span) + self.data
    + self.previous_hash + str(self.nonce)

    return hashlib.sha256(all_block_data.encode()).hexdigest()
```

כעת, כאשר נוסיף בלוק חדש, במקום ישירות לחשב את ה-hash כמו שהיה עד עכשיו כך:

```
new_block.hash = new_block.calculate_hash()
```

נשתמש במנגנון של ה-POW, ונשלח את ה-difficulty הרצוי לנו, למשל 2:

```
new_block.proof_of_work(2)
```

על מנת לעשות זאת גנרי ויפה, נוסיף את התכונה difficulty לשרשרת הבלוקים כך:

```
def __init__(self, difficulty):
    self.chain = [create_genesis_block()]
    self.difficulty = difficulty
```

ואז הפונקציה של הוספת הבלוק תראה:

```
def add_block(self, new_block):
    new_block.previous_hash = self.get_last_block().hash
    # because we change the previous hash then the current hash need to
be calculate again
    new_block.proof_of_work(self.difficulty)
    self.chain.append(new_block)
```

כעת, כאשר ניצור את השרשרת נשלח כפרמטר את ה-difficulty הרצוי:

```
bitcoin = BlockChain(2)
```

לשם נוחות, בסוף הפונקציה של ה-POW נוסיף הדפסה של ה-hash על מנת שנוכל לראות מה קורה.

כעת נריץ את השורות הבאות:

```
bitcoin = BlockChain(2)
bitcoin.add_block(Block(1, '02/01/2022', 'amount = 5'))
bitcoin.add_block(Block(2, '03/01/2022', 'amount = 15'))
```

ונקבל כפלט:

```
00a0951df60fe7e7e591269445fb842c2ff5658184368e9d69d2f83cb4cdac12
0048603866d824383adb6bba3ae92f262aac2cb70dff83356f8e070a3af96a3e
```

ניתן לראות שה-hash של שני הבלוקים מתחיל בשני אפסים - כמו שהגדרנו את ה-difficulty. אך, אם תריצו זאת, תראו שזה קורה נורא מהר, ואין בכך שום קושי לחשב את כל הבלוקים העוקבים. אם נרצה להקשות על האקרים ולהאט את זמן יצירת הבלוקים, נבחר ב-difficulty גדול יותר.

תשנו את ה-difficulty להיות 4 ותוכלו לראות שלוקח כמה שניות עד שיודפס לכם כל hash:

```
00008e6bb6f6417db676076c9510e1236b6ee548a35e232c61e20354ff10a5c7
00000d2681f9840054071a742664088dcc849d1e80d3b4557aca5719a2275129
```

מנגנון זה מאט את יצירת הבלוקים ומקשה על האקרים לחשב את כל הבלוקים העוקבים לבלוק שבו רוצים לחבל. כך נוכל להחליט באיזו מהירות אנו רוצים שיוכלו להוסיף בלוקים ל-blockchain שלנו.

כל הקוד שכתבנו:

```
import hashlib

class Block:
    def __init__(self, index, time_span, data, previous_hash=''):
        self.index = index  # index in the chain
        self.time_span = time_span  # created time
        self.data = data  # some data
        self.previous_hash = previous_hash  # hash of the previous block
                                            # in the chain
        self.nonce = 0
        self.hash = self.calculate_hash()  # current hash

    def calculate_hash(self):
        all_block_data = str(self.index) + str(self.time_span)
```

```
        + self.data + self.previous_hash + str(self.nonce)
        return hashlib.sha256(all_block_data.encode()).hexdigest()

    def proof_of_work(self, difficulty):
        while self.hash[:difficulty] != ''.zfill(difficulty):
            self.nonce += 1
            self.hash = self.calculate_hash()
        print(self.hash)

    def __str__(self):
        return 'block hash: ' + self.hash + '\n     at index: ' + \
               str(self.index) + ' created at: ' + self.time_span + \
               ' data: ' + self.data + '\n     previous hash: ' + \
               self.previous_hash


def create_genesis_block():
    return Block(0, '01/01/2022', 'Genesis block', '0')


class BlockChain:
    def __init__(self, difficulty):
        self.chain = [create_genesis_block()]
        self.difficulty = difficulty

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

    def add_block(self, new_block):
        new_block.previous_hash = self.get_last_block().hash
        # because we change the previous hash then the current
        #hash need to be calculate again
        new_block.proof_of_work(self.difficulty)  # instead of:
        # new_block.hash = new_block.calculate_hash()
        self.chain.append(new_block)

    def is_chain_valid(self):
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i-1]
            if current_block.hash != current_block.calculate_hash() or
               current_block.previous_hash != previous_block.hash:
                return False
        return True

    def __str__(self):
        chain = ''
        for block in self.chain:
            chain += str(block) + '\n'
        return chain


bitcoin = BlockChain(4)
bitcoin.add_block(Block(1, '02/01/2022', 'amount = 5'))
bitcoin.add_block(Block(2, '03/01/2022', 'amount = 15'))
```

תריצו בעצמכם ותראו כמה זמן לוקח למחשב שלכם לחשב זאת. (תוכלו גם לנסות מספרים גדולים יותר).

סיכום

לסיכום, במאמר זה סרקתי את עקרונות טכנולוגית ה-blockchain וכן מימשתי את העקרונות הבסיסיים בפייתון. רבים הגופים המשתמשים בטכנולוגיה זו בזכות יתרונותיה, ונראה שעתידה לגדול. מלבד השימוש היעיל במטבעות דיגיטליים, הטכנולוגיה צפויה לשנות את העולם העסקי ונראה שתוביל למהפכה בעולם הפיננסי.

קישור לקריאה נוספת

- https://builtin.com/blockchain
- https://www.investopedia.com/terms/b/blockchain.asp

הקשר בין Bitcoin ו-Blockchain:

- https://www.pwc.com/us/en/industries/financial-services/fintech/bitcoin-blockchain-cryptocurrency.html

שימוש ב-blockchain ב-Smart Contract:

- https://he.wikipedia.org/wiki/%D7%97%D7%95%D7%96%D7%94_%D7%97%D7%9B%D7%9D

הרחבה על Proof Of Work:

- https://en.wikipedia.org/wiki/Proof_of_work