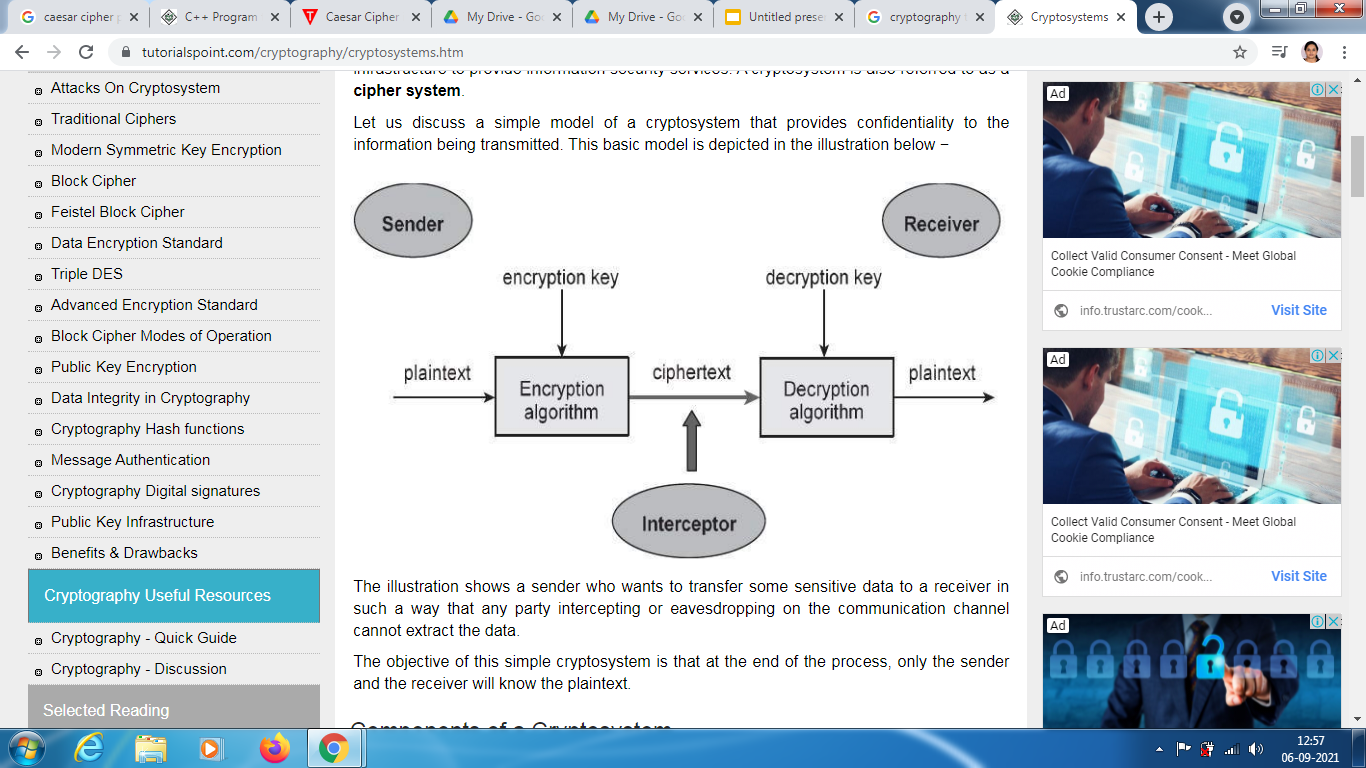
**CRYPTOGRAPHY**

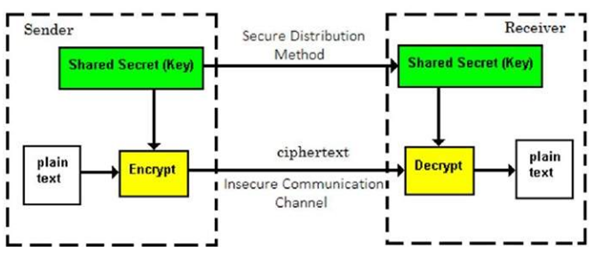
Cryptography is the art and science of making a cryptosystem that is capable of providing information security.

Cryptography deals with the actual securing of digital data. It refers to the design of mechanisms based on mathematical algorithms that provide fundamental information security services.



**SYMMETRIC KEY ENCRYPTION**

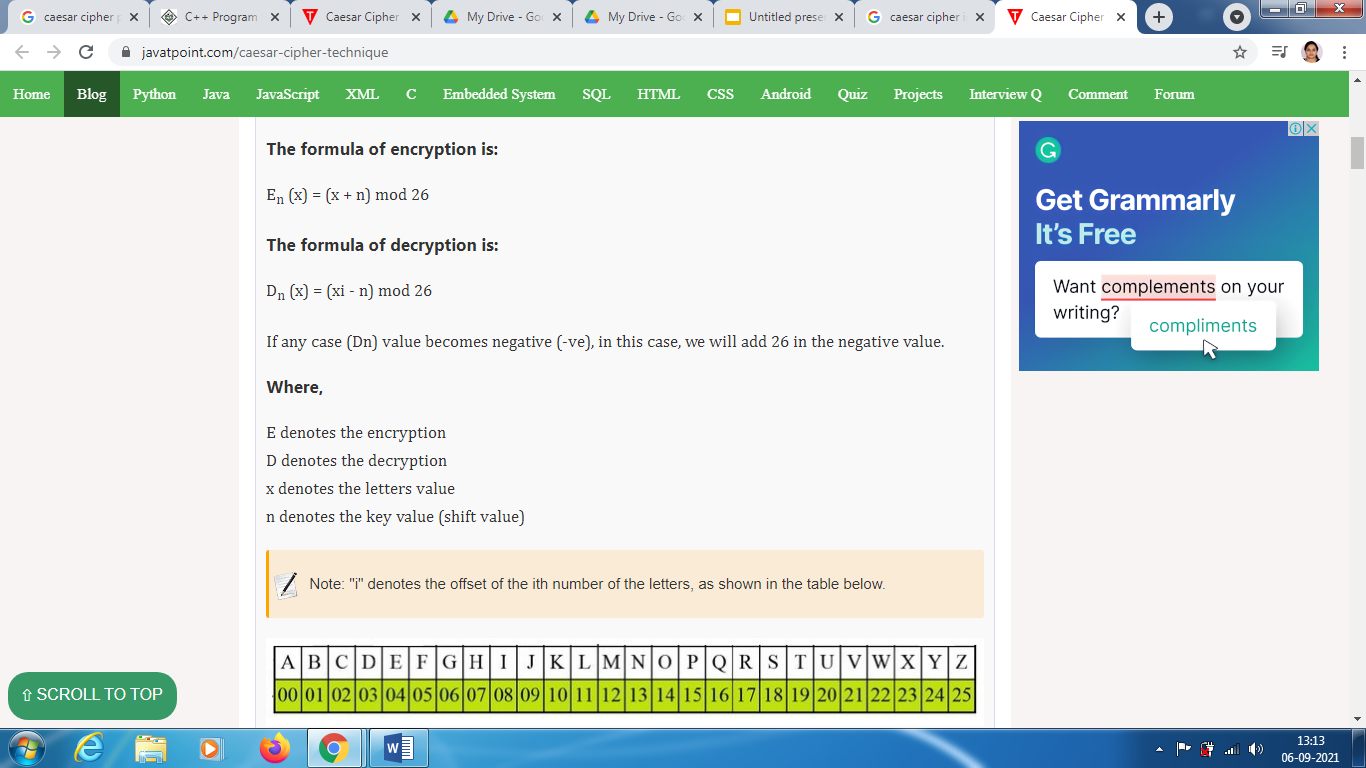
The encryption process where same keys are used for encrypting and decrypting the information is known as Symmetric Key Encryption.



**CAESAR CIPHER TECHNIQUE**

The Caesar cipher is the simplest and oldest method of cryptography. The Caesar cipher method is based on a mono-alphabetic cipher and is also called a shift cipher or additive cipher.

Julius Caesar used the shift cipher (additive cipher) technique to communicate with his officers. For this reason, the shift cipher technique is called the Caesar cipher. The Caesar cipher is a kind of replacement (substitution) cipher, where all letter of plain text is replaced by another letter.



**Example:** 1 Use the Caesar cipher to encrypt and decrypt the message "JAVA," and the key (shift) value of this message is 3.

**Encryption**

We apply encryption formulas by character, based on alphabetical order.

The formula of encryption is:

En (x) = (x + n) mod 26



**Program:**

def encypt\_func(txt, s):

result = ""

# transverse the plain txt

for i in range(len(txt)):

char = txt[i]

# encypt\_func uppercase characters in plain txt

if (char.isupper()):

result += chr((ord(char) + s - 65) % 26 + 65)

# encypt\_func lowercase characters in plain txt

else:

result += chr((ord(char) + s - 97) % 26 + 97)

return result

# check the above function

txt = "CEASER CIPHER EXAMPLE"

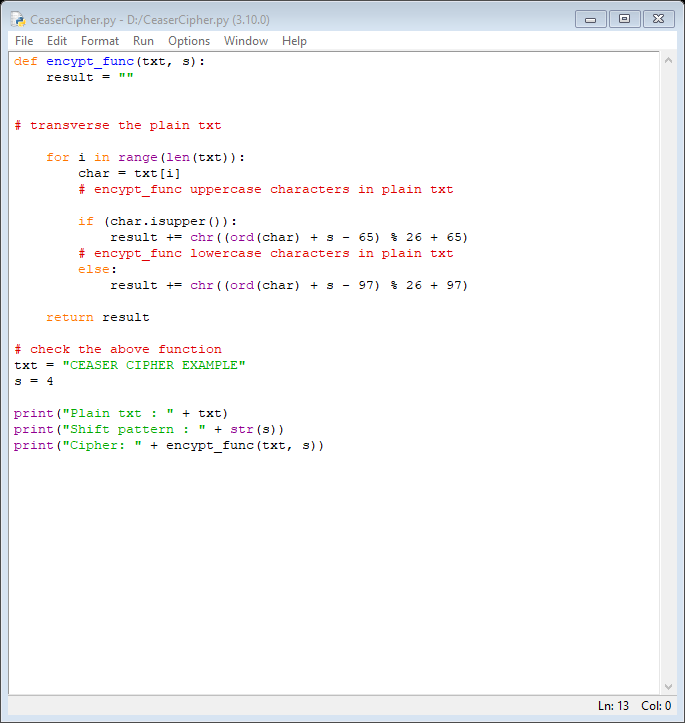
s = 4

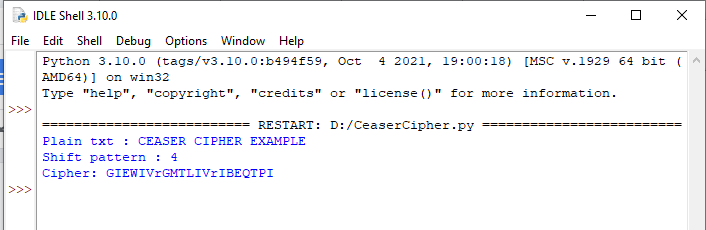
print("Plain txt : " + txt)

print("Shift pattern : " + str(s))

print("Cipher: " + encypt\_func(txt, s))

**Output:**

****

****

**RSA ENCRYPTION ALGORITHM**

RSA encryption algorithm is a type of public-key encryption algorith…SO first we will understand public key.

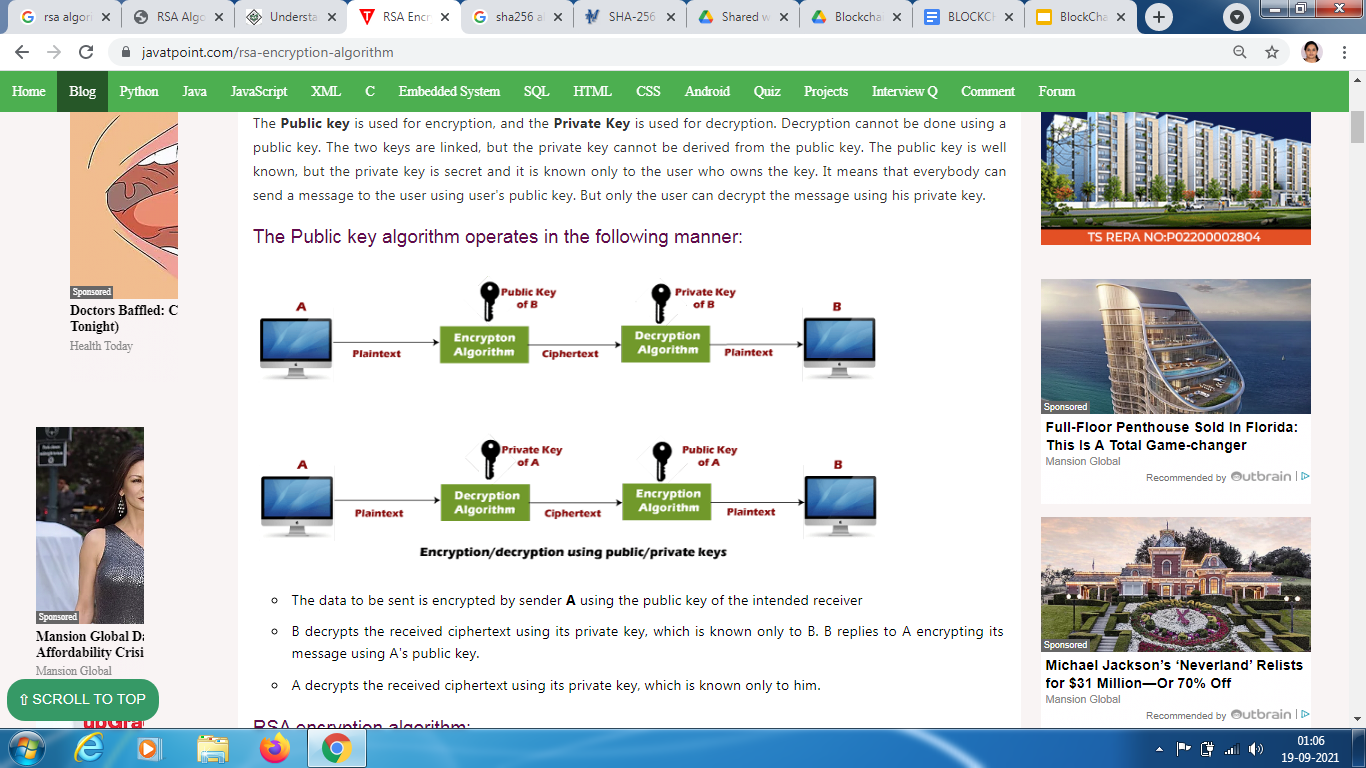
Public Key encryption algorithm Public key encryption algorithm:

is also called the Asymmetric algorithm. Asymmetric algorithms are those algorithms in which sender and receiver use different keys for encryption and decryption. Each sender is assigned a pair of keys:

* Public key
* Private key

The Public key is used for encryption, and the Private Key is used for decryption. Decryption cannot be done using a public key. The two keys are linked, but the private key cannot be derived from the public key.

The public key is well known, but the private key is secret and it is known only to the user who owns the key. It means that everybody can send a message to the user using user's public key. But only the user can decrypt the message using his private key.



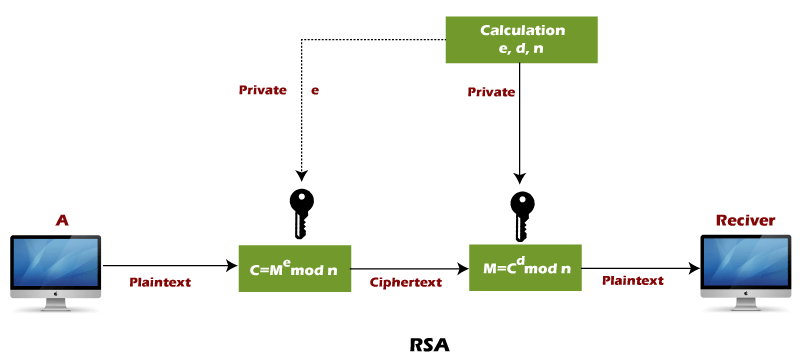
The data to be sent is encrypted by sender A using the public key of the intended receiver

B decrypts the received ciphertext using its private key, which is known only to B. B replies to A encrypting its message using A's public key.

A decrypts the received ciphertext using its private key, which is known only to him.

**RSA encryption algorithm:**

RSA is the most common public-key algorithm, named after its inventors Rivest, Shamir, and Adelman (RSA).



**RSA algorithm uses the following procedure to generate public and private keys:**

* Select two large prime numbers, p and q.
* Multiply these numbers to find n = p x q, where n is called the modulus for encryption and decryption.
* Choose a number e less than n, such that n is relatively prime to (Euler totient function/ Euler phi function ) φ (n)=(p - 1) x (q -1). It means that e and (p - 1) x (q - 1) have no common factor except 1.
* Choose "e" such that 1<e < φ (n), e is prime to φ (n), gcd (e,d(n)) =1
* If n = p x q, then the public key is <e, n>. A plaintext message m is encrypted using public key <e, n>. To find ciphertext from the plain text following formula is used to get ciphertext C.

C = me mod n

Here, m must be less than n. A larger message (>n) is treated as a concatenation of messages, each of which is encrypted separately.

* To determine the private key, we use the following formula to calculate the d such that:

De mod {(p - 1) x (q - 1)} = 1

Or

De mod φ (n) = 1

* The private key is <d, n>. A ciphertext message c is decrypted using private key <d, n>. To calculate plain text m from the ciphertext c following formula is used to get plain text m.

m = cd mod n

**Example:**

This example shows how we can encrypt plaintext 9 using the RSA public-key encryption algorithm. This example uses prime numbers 7 and 11 to generate the public and private keys.

**Explanation:**

Step 1: Select two large prime numbers, p, and q.

p = 7

q = 11

Step 2: Multiply these numbers to find n = p x q, where n is called the modulus for encryption and decryption.

First, we calculate

n = p x q

n = 7 x 11

n = 77

Step 3: Choose a number e less that n, such that n is relatively prime to (p - 1) x (q -1). It means that e and (p - 1) x (q - 1) have no common factor except 1. Choose "e" such that 1<e < φ (n), e is prime to φ (n), gcd (e, d (n)) =1.

Second, we calculate

φ (n) = (p - 1) x (q-1)

φ (n) = (7 - 1) x (11 - 1)

φ (n) = 6 x 10

φ (n) = 60

Let us now choose relative prime e of 60 as 7.

Thus the public key is <e, n> = (7, 77)

Step 4: A plaintext message m is encrypted using public key <e, n>. To find ciphertext from the plain text following formula is used to get ciphertext C.

To find ciphertext from the plain text following formula is used to get ciphertext C.

C = me mod n

C = 97 mod 77

C = 37

Step 5: The private key is <d, n>. To determine the private key, we use the following formula d such that:

De mod {(p - 1) x (q - 1)} = 1

7d mod 60 = 1, which gives d = 43

The private key is <d, n> = (43, 77)

Step 6: A ciphertext message c is decrypted using private key <d, n>. To calculate plain text m from the ciphertext c following formula is used to get plain text m.

m = cd mod n

m = 3743 mod 77

m = 9

In this example, Plain text = 9 and the ciphertext = 37

**Program:**

try:

input = raw\_input

except NameError:

pass

try:

chr = unichr

except NameError:

pass

p=int(input('Enter prime p: '))

q=int(input('Enter prime q: '))

print("Choosen primes:\np=" + str(p) + ", q=" + str(q) + "\n")

n=p\*q

print("n = p \* q = " + str(n) + "\n")

phi=(p-1)\*(q-1)

print("Euler's function (totient) [phi(n)]: " + str(phi) + "\n")

def gcd(a, b):

while b != 0:

c = a % b

a = b

b = c

return a

def modinv(a, m):

for x in range(1, m):

if (a \* x) % m == 1:

return x

return None

def coprimes(a):

l = []

for x in range(2, a):

if gcd(a, x) == 1 and modinv(x,phi) != None:

l.append(x)

for x in l:

if x == modinv(x,phi):

l.remove(x)

return l

print("Choose an e from a below coprimes array:\n")

print(str(coprimes(phi)) + "\n")

e=int(input())

d=modinv(e,phi)

print("\nYour public key is a pair of numbers (e=" + str(e) + ", n=" + str(n) + ").\n")

print("Your private key is a pair of numbers (d=" + str(d) + ", n=" + str(n) + ").\n")

def encrypt\_block(m):

c = modinv(m\*\*e, n)

if c == None: print('No modular multiplicative inverse for block ' + str(m) + '.')

return c

def decrypt\_block(c):

m = modinv(c\*\*d, n)

if m == None: print('No modular multiplicative inverse for block ' + str(c) + '.')

return m

def encrypt\_string(s):

return ''.join([chr(encrypt\_block(ord(x))) for x in list(s)])

def decrypt\_string(s):

return ''.join([chr(decrypt\_block(ord(x))) for x in list(s)])

s = input("Enter a message to encrypt: ")

print("\nPlain message: " + s + "\n")

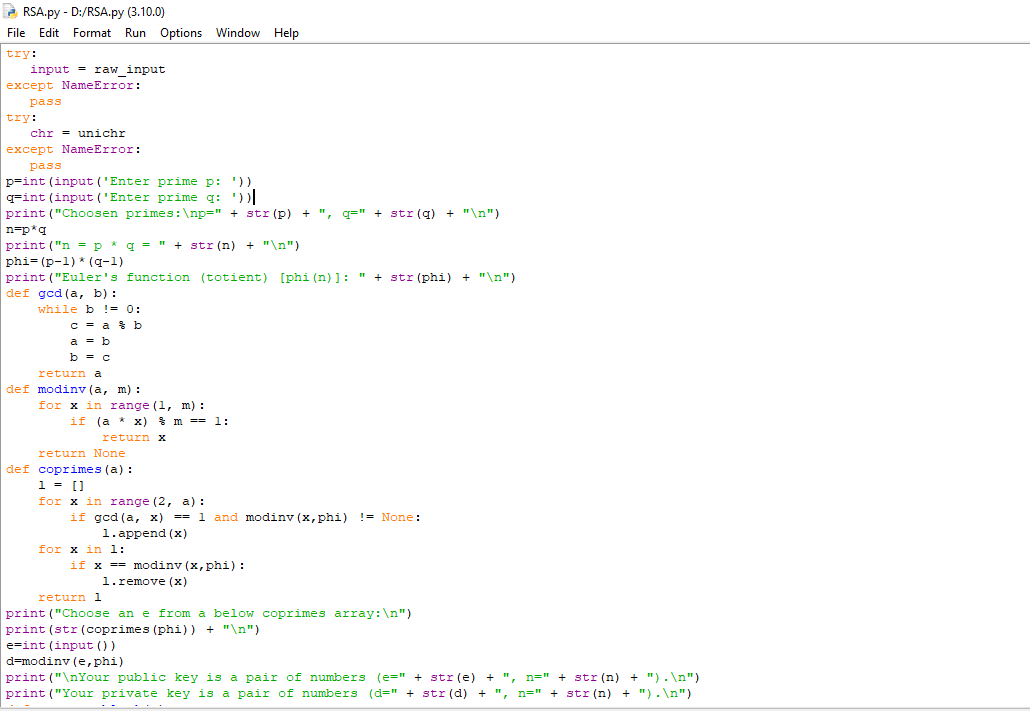
enc = encrypt\_string(s)

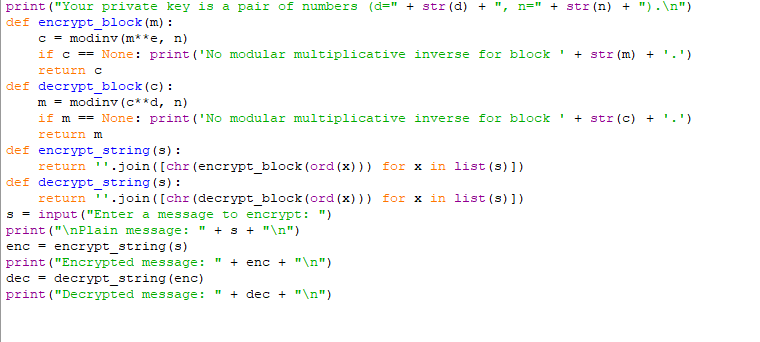
print("Encrypted message: " + enc + "\n")

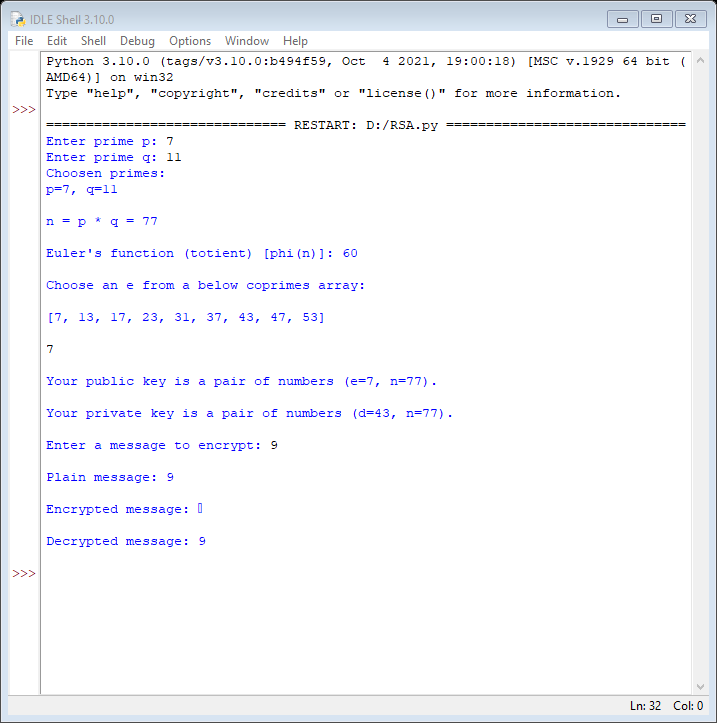
dec = decrypt\_string(enc)

print("Decrypted message: " + dec + "\n")

**Output:**

****

****

****

**HASH FUNCTION**

Three of the main purposes of a hash function are:

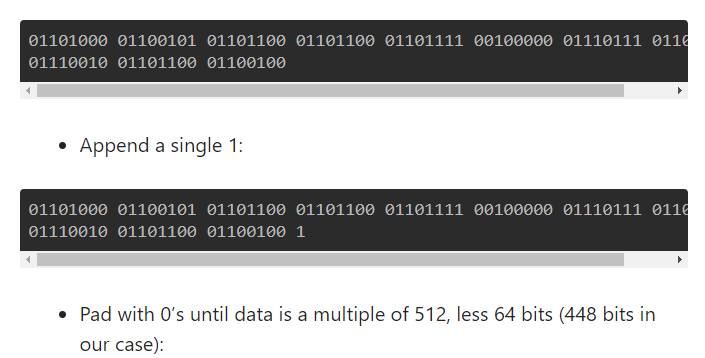
* To scramble data deterministically
* To accept an input of arbitrary length and output a fixed length result
* To manipulate data irreversibly. The input cannot be derived from the output

SHA-2 is an algorithm, a generalized idea of how to hash data. SHA-2 has several variants, all of which use the same algorithm but use different constants. SHA-256, for example, sets additional constants that define the behavior of the SHA-2 algorithm, one of these constants is the output size, 256. The 256 and 512 in SHA-256 and SHA-512 refer to the respective digest size in bits.

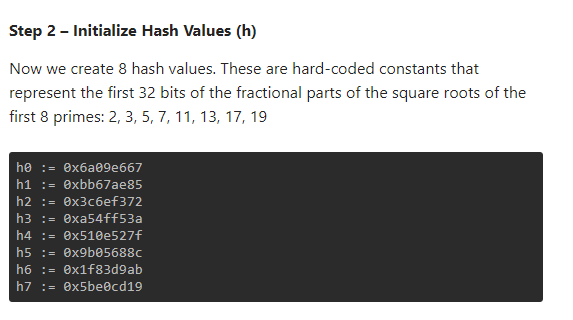
**Step-by-step SHA-256 hash of “hello world”**

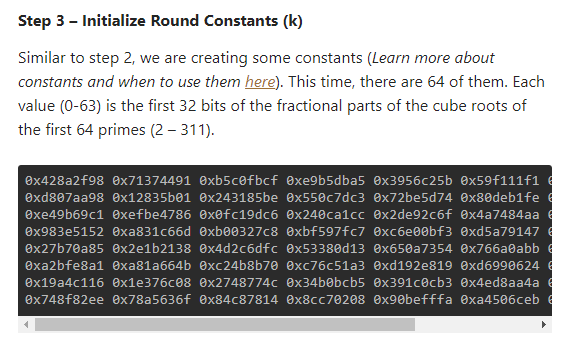
**Step 1 – Pre-Processing**

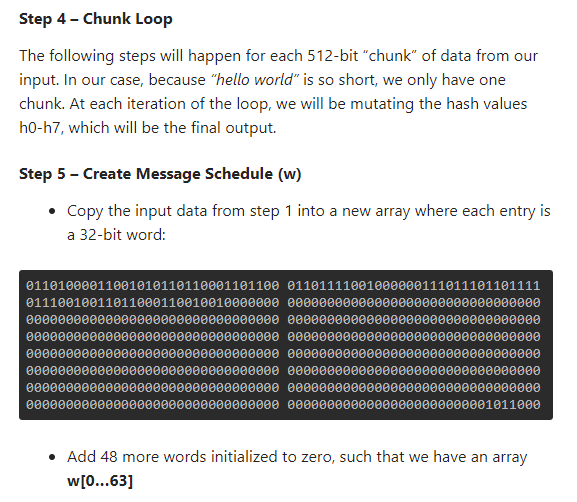
* Convert “hello world” to binary:

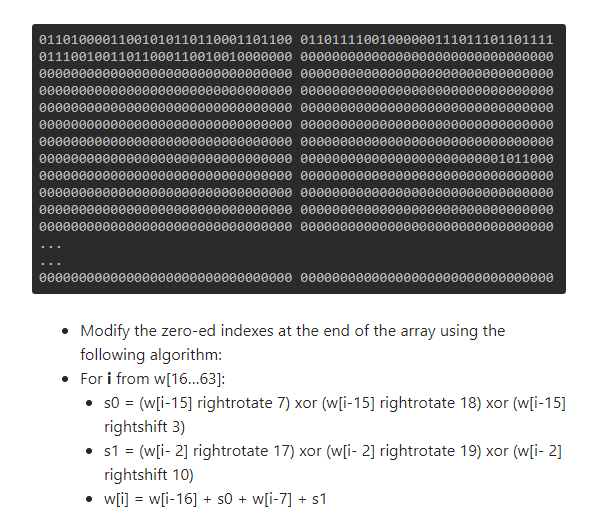


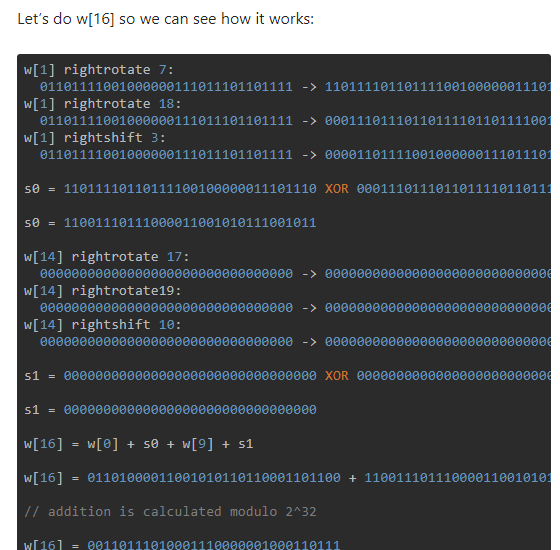


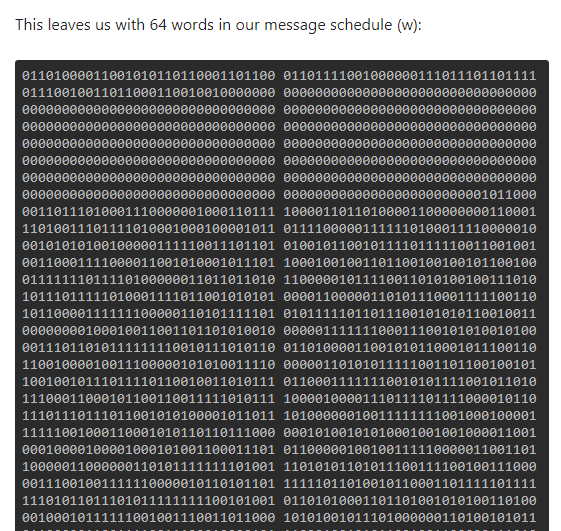


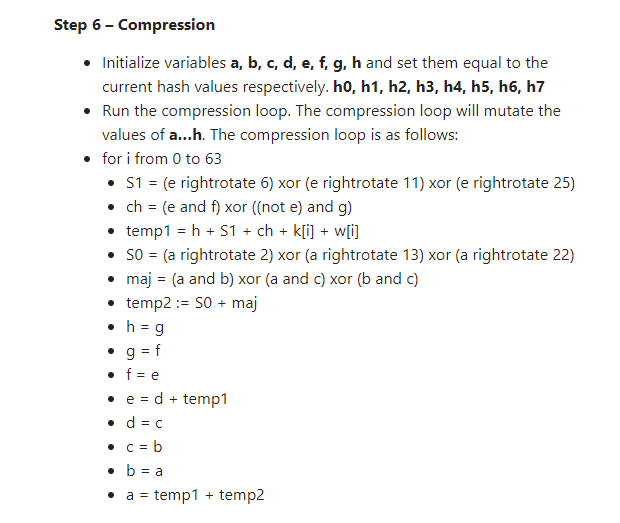




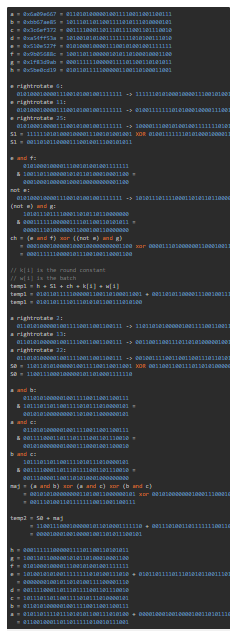


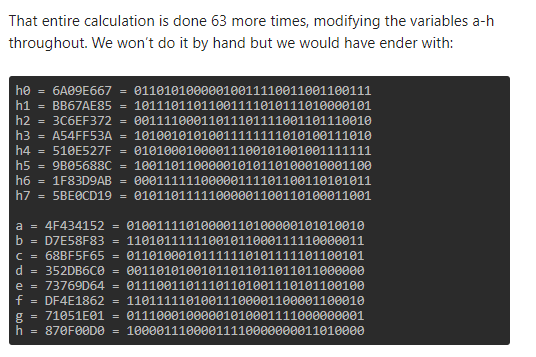


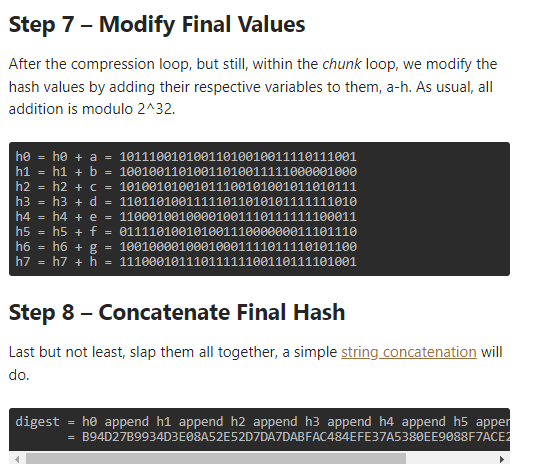




Let’s go through the first iteration, all addition is calculated modulo 2^32







**Program:**

import hashlib

string="Uday2012uuu"

encoded=string.encode()

result = hashlib.sha256(encoded)

print("String : ", end ="")

print(string)

print("Hash Value : ", end ="")

print(result)

print("Hexadecimal equivalent: ",result.hexdigest())

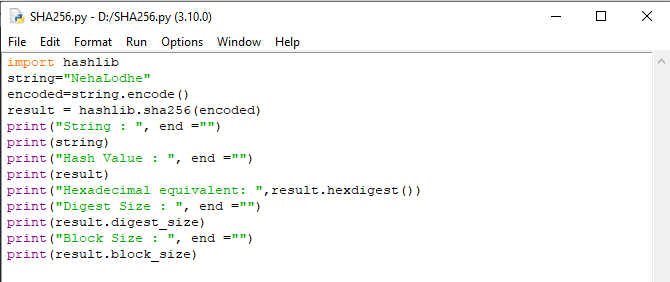
print("Digest Size : ", end ="")

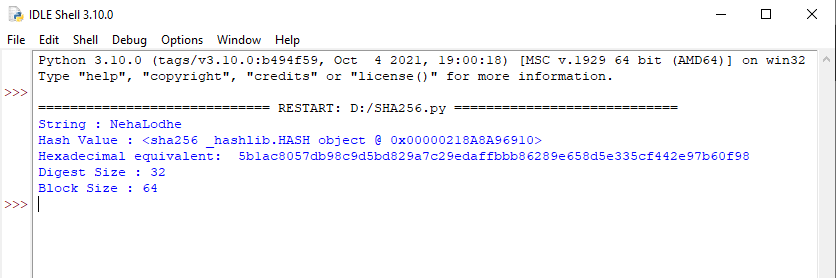
print(result.digest\_size)

print("Block Size : ", end ="")

print(result.block\_size)

**Output:**

****

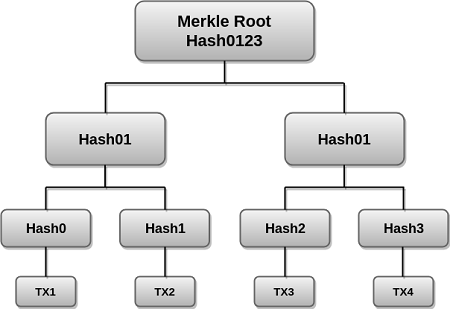
****

**MERKLE TREE**

Merkle tree is a fundamental part of blockchain technology. It is a mathematical data structure composed of hashes of different blocks of data, and which serves as a summary of all the transactions in a block. It also allows for efficient and secure verification of content in a large body of data. It also helps to verify the consistency and content of the data. Both Bitcoin and Ethereum use Merkle Trees structure. Merkle Tree is also known as Hash Tree.

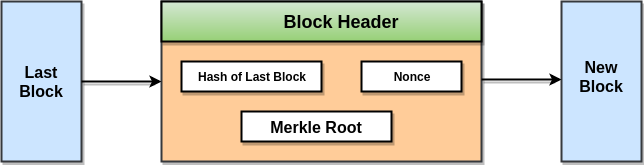
The concept of Merkle Tree is named after Ralph Merkle, who patented the idea in 1979. Fundamentally, it is a data structure tree in which every leaf node labelled with the hash of a data block, and the non-leaf node labelled with the cryptographic hash of the labels of its child nodes. The leaf nodes are the lowest node in the tree.

Every leaf node is a hash of transactional data, and the non-leaf node is a hash of its previous hashes. Merkle trees are in a binary tree, so it requires an even number of leaf nodes. If there is an odd number of transactions, the last hash will be duplicated once to create an even number of leaf nodes.



The above example is the most common and simple form of a Merkle tree, i.e., **Binary Merkle Tree**. There are four transactions in a block: **TX1**, **TX2**, **TX3**, and **TX4**. Here you can see, there is a top hash which is the hash of the entire tree, known as the **Root Hash**, or the **Merkle Root**. Each of these is repeatedly hashed, and stored in each leaf node, resulting in Hash 0, 1, 2, and 3. Consecutive pairs of leaf nodes are then summarized in a parent node by hashing **Hash0** and **Hash1**, resulting in **Hash01**, and separately hashing **Hash2** and **Hash3**, resulting in **Hash23**. The two hashes (**Hash01** and **Hash23**) are then hashed again to produce the Root Hash or the Merkle Root.

Merkle Root is stored in the **block header**. The block header is the part of the bitcoin block which gets hash in the process of mining. It contains the hash of the last block, a Nonce, and the Root Hash of all the transactions in the current block in a Merkle Tree. So having the Merkle root in block header makes the transaction **tamper-proof**. As this Root Hash includes the hashes of all the transactions within the block, these transactions may result in saving the disk space.



The Merkle Tree maintains the **integrity** of the data. If any single detail of transactions or order of the transaction's changes, then these changes reflected in the hash of that transaction. This change would cascade up the Merkle Tree to the Merkle Root, changing the value of the Merkle root and thus invalidating the block. So everyone can see that Merkle tree allows for a quick and simple test of whether a specific transaction is included in the set or not.

**Merkle trees have three benefits:**

* It provides a means to maintain the integrity and validity of data.
* It helps in saving the memory or disk space as the proofs,

computationally easy and fast.

* Their proofs and management require tiny amounts of information

to be transmitted across networks.

**Program:**

from hashlib import sha256

class MerkleNode:

""" Stores the hash and the parent. """

def \_\_init\_\_(self, hash):

self.hash = hash

self.parent = None

class MerkleTree:

""" Stores the leaves and the root hash of the tree. """

def \_\_init\_\_(self, data\_chunks):

leaves = []

for chunk in data\_chunks:

node = MerkleNode(self.compute\_hash(chunk))

leaves.append(node)

self.root = self.build\_merkle\_tree(leaves)

def build\_merkle\_tree(self, leaves):

"""

Builds the Merkle tree from a list of leaves. In case of an odd number of leaves, the last leaf is duplicated.

"""

num\_leaves = len(leaves)

if num\_leaves == 1:

return leaves[0]

parents = []

i = 0

while i < num\_leaves:

left\_child = leaves[i]

right\_child = leaves[i + 1] if i + 1 < num\_leaves else left\_child

parents.append(self.create\_parent(left\_child, right\_child))

i += 2

return self.build\_merkle\_tree(parents)

def create\_parent(self, left\_child, right\_child):

"""

Creates the parent node from the children, and updates

their parent field.

"""

parent = MerkleNode(

self.compute\_hash(left\_child.hash + right\_child.hash))

left\_child.parent, right\_child.parent = parent, parent

print("Left child: {}, Right child: {}, Parent: {}".format(

left\_child.hash, right\_child.hash, parent.hash))

return parent

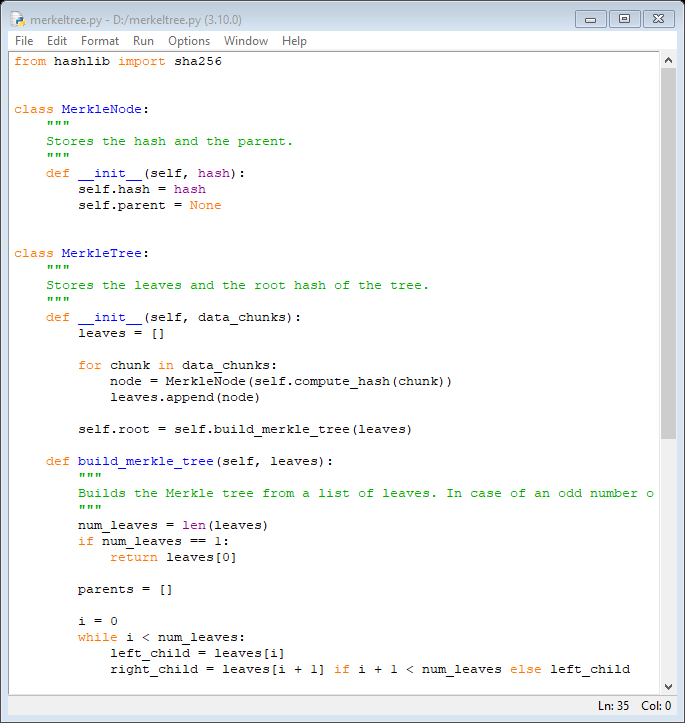
@staticmethod

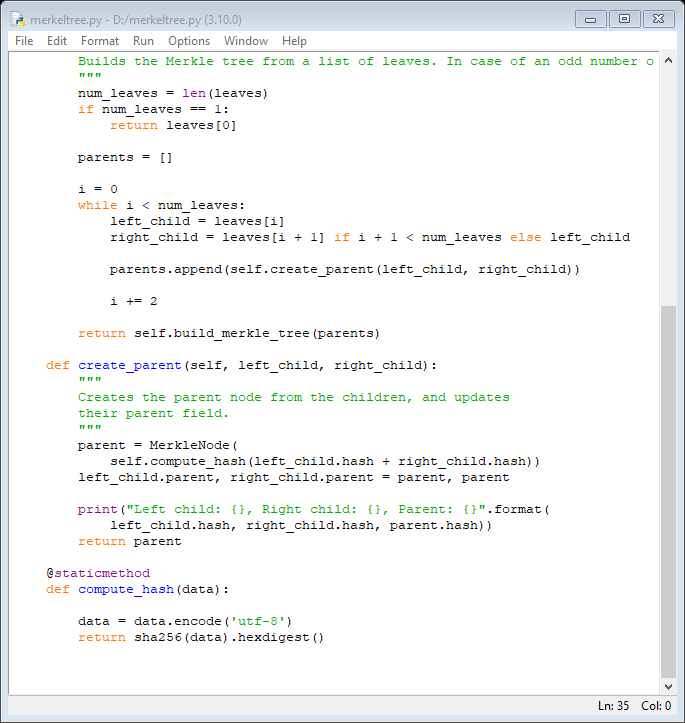
def compute\_hash(data):

data = data.encode('utf-8')

return sha256(data).hexdigest()

**Output:**

****

****

(For Output execute these commands )

file = "hello"

>>> chunks = list(file)

>>> chunks

['h', 'e', 'l', 'l', 'o']

>>> merkle\_tree = MerkleTree(chunks)

>>> print(merkle\_tree.root.hash)

