# Cryptography and Network Security Lab

## PRN: 2020BTECS00064

**Name: Pranit Sankpal Batch: B4**

**ADVANCED ENCRYPTION STANDARD ALGORITHM**

**Aim:**

To encrypt and decrypt the plain text using AES.

**Theory:**

The more popular and widely adopted symmetric encryption algorithm likely to be encountered nowadays is the Advanced Encryption Standard (AES). It is found at least six time faster than triple DES.

A replacement for DES was needed as its key size was too small. With increasing computing power, it was considered vulnerable against exhaustive key search attack. Triple DES was designed to overcome this drawback but it was found slow.

The features of AES are as follows −

* Symmetric key symmetric block cipher
* 128-bit data, 128/192/256-bit keys
* Stronger and faster than Triple-DES
* Provide full specification and design details
* Software implementable in C and Java

## Operation of AES

AES is an iterative rather than Feistel cipher. It is based on ‘substitution–permutation network’. It comprises of a series of linked operations, some of which involve replacing inputs by specific outputs (substitutions) and others involve shuffling bits around (permutations).

Interestingly, AES performs all its computations on bytes rather than bits. Hence, AES treats the 128 bits of a plaintext block as 16 bytes. These 16 bytes are arranged in four columns and four rows for processing as a matrix −

Unlike DES, the number of rounds in AES is variable and depends on the length of the key. AES uses 10 rounds for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit keys. Each of these rounds uses a different 128-bit round key, which is calculated from the original AES key.

The schematic of AES structure is given in the following illustration −



## Encryption Process

Here, we restrict to description of a typical round of AES encryption. Each round comprise of four sub-processes. The first round process is depicted below −



### **Byte Substitution (SubBytes)**

The 16 input bytes are substituted by looking up a fixed table (S-box) given in design. The result is in a matrix of four rows and four columns.

### **Shiftrows**

Each of the four rows of the matrix is shifted to the left. Any entries that ‘fall off’ are re-inserted on the right side of row. Shift is carried out as follows −

* First row is not shifted.
* Second row is shifted one (byte) position to the left.
* Third row is shifted two positions to the left.
* Fourth row is shifted three positions to the left.
* The result is a new matrix consisting of the same 16 bytes but shifted with respect to each other.

### **MixColumns**

Each column of four bytes is now transformed using a special mathematical function. This function takes as input the four bytes of one column and outputs four completely new bytes, which replace the original column. The result is another new matrix consisting of 16 new bytes. It should be noted that this step is not performed in the last round.

### **Addroundkey**

The 16 bytes of the matrix are now considered as 128 bits and are XORed to the 128 bits of the round key. If this is the last round then the output is the ciphertext. Otherwise, the resulting 128 bits are interpreted as 16 bytes and we begin another similar round.

## Decryption Process

The process of decryption of an AES ciphertext is similar to the encryption process in the reverse order. Each round consists of the four processes conducted in the reverse order −

* Add round key
* Mix columns
* Shift rows
* Byte substitution

Since sub-processes in each round are in reverse manner, unlike for a Feistel Cipher, the encryption and decryption algorithms needs to be separately implemented, although they are very closely related.

**Code:**

from block\_cipher import BlockCipher, BlockCipherWrapper

from block\_cipher import MODE\_ECB, MODE\_CBC, MODE\_CFB, MODE\_OFB, MODE\_CTR

def print\_str\_into\_AES\_matrix(str):

    characters = ' '.join([str[i:i+2] for i in range(0, len(str), 2)]).split()

    matrix = [[characters[i] for i in range(j, j + 4)] for j in range(0, len(characters), 4)]

    for col in range(4):

        for row in range(4):

            print(matrix[row][col], end=" ")

        print()

\_\_all\_\_ = [

    'new', 'block\_size', 'key\_size',

    'MODE\_ECB', 'MODE\_CBC', 'MODE\_CFB', 'MODE\_OFB', 'MODE\_CTR'

]

SBOX = (

    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,

    0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,

    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,

    0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,

    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,

    0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,

    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,

    0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,

    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,

    0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,

    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,

    0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,

    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,

    0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,

    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,

    0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,

    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17,

    0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,

    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88,

    0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,

    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,

    0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,

    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,

    0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,

    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6,

    0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,

    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,

    0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,

    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,

    0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,

    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,

    0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16,

)

INV\_SBOX = (

    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38,

    0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,

    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87,

    0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,

    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d,

    0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,

    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2,

    0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,

    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16,

    0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,

    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda,

    0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,

    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a,

    0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,

    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02,

    0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,

    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea,

    0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,

    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85,

    0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,

    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89,

    0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,

    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20,

    0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,

    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31,

    0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,

    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d,

    0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,

    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0,

    0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,

    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26,

    0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d,

)

round\_constants = (0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36)

block\_size = 16

key\_size = None

def new(key, mode, IV=None, \*\*kwargs) -> BlockCipherWrapper:

    if mode in (MODE\_CBC, MODE\_CFB, MODE\_OFB) and IV is None:

        raise ValueError("This mode requires an IV")

    cipher = BlockCipherWrapper()

    cipher.block\_size = block\_size

    cipher.IV = IV

    cipher.mode = mode

    cipher.cipher = AES(key)

    if mode == MODE\_CFB:

        cipher.segment\_size = kwargs.get('segment\_size', block\_size \* 8)

    elif mode == MODE\_CTR:

        counter = kwargs.get('counter')

        if counter is None:

            raise ValueError("CTR mode requires a callable counter object")

        cipher.counter = counter

    return cipher

class AES(BlockCipher):

    def \_\_init\_\_(self, key: bytes):

        self.key = key

        self.Nk = len(self.key) // 4  # words per key

        if self.Nk not in (4, 6, 8):

            raise ValueError("Invalid key size")

        self.Nr = self.Nk + 6

        self.Nb = 4  # words per block

        self.state: list[list[int]] = []

        # raise NotImplementedError

        # key schedule

        self.w: list[list[int]] = []

        for i in range(self.Nk):

            self.w.append(list(key[4\*i:4\*i+4]))

        for i in range(self.Nk, self.Nb\*(self.Nr+1)):

            tmp: list[int] = self.w[i-1]

            q, r = divmod(i, self.Nk)

            if not r:

                tmp = self.sub\_word(self.rot\_word(tmp))

                tmp[0] ^= round\_constants[q-1]

            elif self.Nk > 6 and r == 4:

                tmp = self.sub\_word(tmp)

            self.w.append(

                [a ^ b for a, b in zip(self.w[i-self.Nk], tmp)]

            )

    def encrypt\_block(self, block: bytes) -> bytes:

        self.set\_state(block)

        self.add\_round\_key(0)

        print("\nInitial:")

        print\_str\_into\_AES\_matrix(self.get\_state().hex())

        for r in range(1, self.Nr):

            print(f"\nRound {r}:")

            self.sub\_bytes()

            print("After SubBytes:")

            print\_str\_into\_AES\_matrix(self.get\_state().hex())

            self.shift\_rows()

            print("After ShiftRows:")

            print\_str\_into\_AES\_matrix(self.get\_state().hex())

            self.mix\_columns()

            print("After MixColumns:")

            print\_str\_into\_AES\_matrix(self.get\_state().hex())

            self.add\_round\_key(r)

            print("After AddRoundKey:")

            print\_str\_into\_AES\_matrix(self.get\_state().hex())

        print(f"\nFinal Round {r+1}:")

        self.sub\_bytes()

        print("After SubBytes:")

        print\_str\_into\_AES\_matrix(self.get\_state().hex())

        self.shift\_rows()

        print("After ShiftRows:")

        print\_str\_into\_AES\_matrix(self.get\_state().hex())

        self.add\_round\_key(self.Nr)

        print("After AddRoundKey:")

        print\_str\_into\_AES\_matrix(self.get\_state().hex())

        return self.get\_state()

    def decrypt\_block(self, block: bytes) -> bytes:

        self.set\_state(block)

        print("\nInitial:")

        print\_str\_into\_AES\_matrix(self.get\_state().hex())

        self.add\_round\_key(self.Nr)

        for r in range(self.Nr-1, 0, -1):

            print(f"\nRound {r}:")

            self.inv\_shift\_rows()

            print("After Inverse ShiftRows:")

            print\_str\_into\_AES\_matrix(self.get\_state().hex())

            self.inv\_sub\_bytes()

            print("After Inverse SubBytes:")

            print\_str\_into\_AES\_matrix(self.get\_state().hex())

            self.add\_round\_key(r)

            print("After AddRoundKey:")

            print\_str\_into\_AES\_matrix(self.get\_state().hex())

            self.inv\_mix\_columns()

            print("After Inverse MixColumns:")

            print\_str\_into\_AES\_matrix(self.get\_state().hex())

        print(f"\nFinal Round {r}:")

        self.inv\_shift\_rows()

        print("After Inverse ShiftRows:")

        print\_str\_into\_AES\_matrix(self.get\_state().hex())

        self.inv\_sub\_bytes()

        print("After Inverse SubBytes:")

        print\_str\_into\_AES\_matrix(self.get\_state().hex())

        self.add\_round\_key(0)

        print("After AddRoundKey:")

        print\_str\_into\_AES\_matrix(self.get\_state().hex())

        return self.get\_state()

    @staticmethod

    def rot\_word(word: list[int]):

        # for key schedule

        return word[1:] + word[:1]

    @staticmethod

    def sub\_word(word: list[int]):

        # for key schedule

        return [SBOX[b] for b in word]

    def set\_state(self, block: bytes):

        self.state = [

            list(block[i:i+4])

            for i in range(0, 16, 4)

        ]

    def get\_state(self) -> bytes:

        return b''.join(

            bytes(col)

            for col in self.state

        )

    def add\_round\_key(self, r: int):

        round\_key = self.w[r\*self.Nb:(r+1)\*self.Nb]

        for col, word in zip(self.state, round\_key):

            for row\_index in range(4):

                col[row\_index] ^= word[row\_index]

    def mix\_columns(self):

        for i, word in enumerate(self.state):

            new\_word = []

            for j in range(4):

                # element wise cl mul with constants 2, 3, 1, 1

                value = (word[0] << 1)

                value ^= (word[1] << 1) ^ word[1]

                value ^= word[2] ^ word[3]

                # polynomial reduction in constant time

                value ^= 0x11b & -(value >> 8)

                new\_word.append(value)

                # rotate word in order to match the matrix multiplication

                word = self.rot\_word(word)

            self.state[i] = new\_word

    def inv\_mix\_columns(self):

        for i, word in enumerate(self.state):

            new\_word = []

            for j in range(4):

                # element wise cl mul with constants 0xe, 0xb, 0xd, 0x9

                value = (word[0] << 3) ^ (word[0] << 2) ^ (word[0] << 1)

                value ^= (word[1] << 3) ^ (word[1] << 1) ^ word[1]

                value ^= (word[2] << 3) ^ (word[2] << 2) ^ word[2]

                value ^= (word[3] << 3) ^ word[3]

                # polynomial reduction in constant time

                value ^= (0x11b << 2) & -(value >> 10)

                value ^= (0x11b << 1) & -(value >> 9)

                value ^= 0x11b & -(value >> 8)

                new\_word.append(value)

                # rotate word in order to match the matrix multiplication

                word = self.rot\_word(word)

            self.state[i] = new\_word

    def shift\_rows(self):

        for row\_index in range(4):

            row = [

                col[row\_index] for col in self.state

            ]

            row = row[row\_index:] + row[:row\_index]

            for col\_index in range(4):

                self.state[col\_index][row\_index] = row[col\_index]

    def inv\_shift\_rows(self):

        for row\_index in range(4):

            row = [

                col[row\_index] for col in self.state

            ]

            row = row[-row\_index:] + row[:-row\_index]

            for col\_index in range(4):

                self.state[col\_index][row\_index] = row[col\_index]

    def sub\_bytes(self):

        for col in self.state:

            for row\_index in range(4):

                col[row\_index] = SBOX[col[row\_index]]

    def inv\_sub\_bytes(self):

        for col in self.state:

            for row\_index in range(4):

                col[row\_index] = INV\_SBOX[col[row\_index]]

    def print\_state(self):

        # debug function

        for row\_index in range(4):

            print(' '.join(f'{col[row\_index]:02x}' for col in self.state))

        print()

# Main Code

ch = int(input("What do you want to perform?\n1. Encryption\n2. Decryption\n"))

if(ch == 1):

    msg = str(input("Enter the message to be encrypted(16 characters only):\n"))

    if(len(msg) != 16):

        print("Invalid Message size!")

        exit()

    key = str(input("Enter the key for encryption(16 or 24 or 32 characters):\n"))

    key\_length = len(key)

    if (key\_length!=16 and key\_length!=24 and key\_length!=32):

        print("Invalid Key size!")

        exit()

    mode = int(input("Choose the Mode of Operation:\n1. ECB\n2. CBC\n3. CFB\n4. OFB\n5. CTR\n"))

    iv = None

    if mode == 1:

        AES\_MODE = MODE\_ECB

    elif mode == 2:

        AES\_MODE = MODE\_CBC

        iv = str(input("Enter the Initialization Vector(IV) [16 characters]:\n"))

        if(len(iv) != 16):

            print("Invalid IV size!")

            exit()

    elif mode == 3:

        AES\_MODE = MODE\_CFB

        iv = str(input("Enter the Initialization Vector(IV) [16 characters]:\n"))

        if(len(iv) != 16):

            print("Invalid IV size!")

            exit()

    elif mode == 4:

        AES\_MODE = MODE\_OFB

        iv = str(input("Enter the Initialization Vector(IV) [16 characters]:\n"))

        if(len(iv) != 16):

            print("Invalid IV size!")

            exit()

    elif mode == 5:

        AES\_MODE = MODE\_CTR

    else:

        print("Invalid choice !!")

        exit()

    key = bytes.fromhex(key.encode('utf-8').hex())

    plain\_text = bytes.fromhex(msg.encode('utf-8').hex())

    if iv is not None:

        iv = bytes.fromhex(iv.encode('utf-8').hex())

    cipher = new(key, AES\_MODE, IV=iv)

    cipher\_text = cipher.encrypt(plain\_text)

    print(f"\nCiphertext is: {cipher\_text.hex()}")

elif(ch == 2):

    c\_txt = str(input("Enter the ciphertext to be decrypted(16 characters) [in hex format]:\n"))

    if(len(c\_txt) != 32):

        print("Invalid Cipher text size!")

        exit()

    key = str(input("Enter the key for decryption(16 or 24 or 32 characters):\n"))

    key = bytes.fromhex(key.encode('utf-8').hex())

    key\_length = len(key)

    if (key\_length!=16 and key\_length!=24 and key\_length!=32):

        print("Invalid Key size!")

        exit()

    mode = int(input("Choose the Mode of Operation used:\n1. ECB\n2. CBC\n3. CFB\n4. OFB\n5. CTR\n"))

    iv = None

    if mode == 1:

        AES\_MODE = MODE\_ECB

    elif mode == 2:

        AES\_MODE = MODE\_CBC

        iv = str(input("Enter the Initialization Vector(IV) [16 characters]:\n"))

        if(len(iv) != 16):

            print("Invalid IV size!")

            exit()

    elif mode == 3:

        AES\_MODE = MODE\_CFB

        iv = str(input("Enter the Initialization Vector(IV) [16 characters]:\n"))

        if(len(iv) != 16):

            print("Invalid IV size!")

            exit()

    elif mode == 4:

        AES\_MODE = MODE\_OFB

        iv = str(input("Enter the Initialization Vector(IV) [16 characters]:\n"))

        if(len(iv) != 16):

            print("Invalid IV size!")

            exit()

    elif mode == 5:

        AES\_MODE = MODE\_CTR

    else:

        print("Invalid choice !!")

        exit()

    if iv is not None:

        iv = bytes.fromhex(iv.encode('utf-8').hex())

    cipher = new(key, AES\_MODE, IV=iv)

    dec\_bytes = cipher.decrypt(bytes.fromhex(c\_txt))

    dec\_txt = dec\_bytes.decode('utf-8')

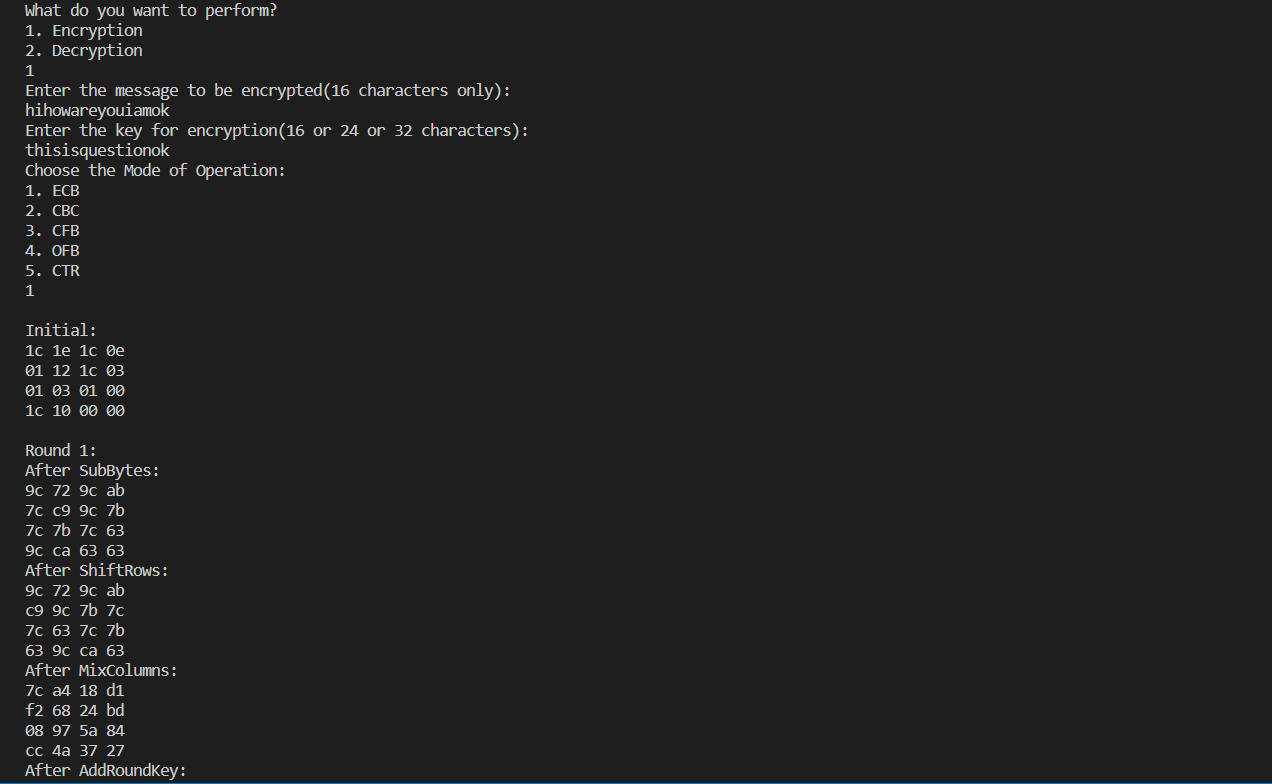
    print(f"\nDecrypted message is: {dec\_txt}")

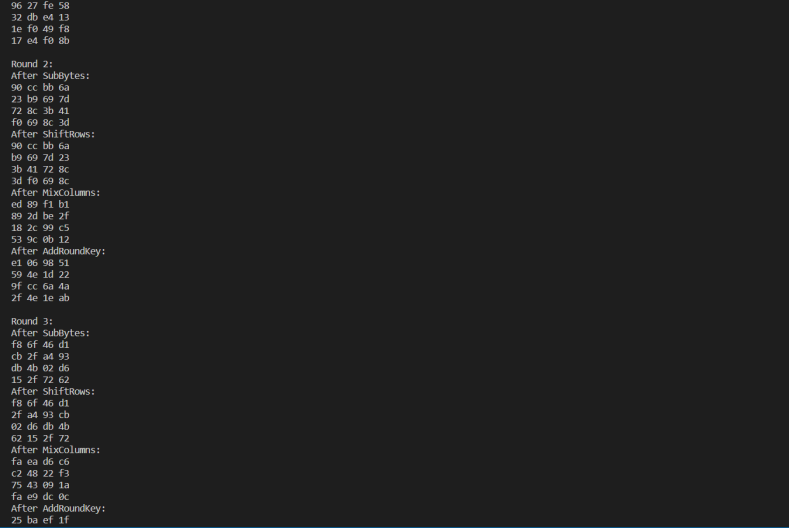
else:

    print("Invalid input!")

**Output:**

**Encryption:**







**Decryption:**

