**Implementation of Advanced Encryption Standard**

In 2001, the National Institute of Standards and Technology (NIST) published Advanced Encryption Standard ( AES). It was created with the intention to replace DES. AES is a symmetric block cipher with a complex structure. AES is a symmetric block cipher with a complex structure, designed to provide a more secure and efficient alternative to DES in modern cryptographic applications.

# **CODE**

from copy import deepcopy

s\_box = [

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\_s\_box = [

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

]

r\_con = [

0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36

]

def bytes\_to\_matrix(text):

return [list(text[i:i+4]) for i in range(0, len(text), 4)]

def matrix\_to\_bytes(matrix):

return bytes(sum(matrix, []))

def add\_round\_key(state, key):

for i in range(4):

for j in range(4):

state[i][j] ^= key[i][j]

def sub\_bytes(state):

for i in range(4):

for j in range(4):

state[i][j] = s\_box[state[i][j]]

def inv\_sub\_bytes(state):

for i in range(4):

for j in range(4):

state[i][j] = inv\_s\_box[state[i][j]]

def shift\_rows(state):

state[1][0], state[1][1], state[1][2], state[1][3] = \

state[1][1], state[1][2], state[1][3], state[1][0]

state[2][0], state[2][1], state[2][2], state[2][3] = \

state[2][2], state[2][3], state[2][0], state[2][1]

state[3][0], state[3][1], state[3][2], state[3][3] = \

state[3][3], state[3][0], state[3][1], state[3][2]

def inv\_shift\_rows(state):

state[1][0], state[1][1], state[1][2], state[1][3] = \

state[1][3], state[1][0], state[1][1], state[1][2]

state[2][0], state[2][1], state[2][2], state[2][3] = \

state[2][2], state[2][3], state[2][0], state[2][1]

state[3][0], state[3][1], state[3][2], state[3][3] = \

state[3][1], state[3][2], state[3][3], state[3][0]

def mix\_single\_column(a):

t = a[0] ^ a[1] ^ a[2] ^ a[3]

u = a[0]

a[0] ^= t ^ xtime(a[0] ^ a[1])

a[1] ^= t ^ xtime(a[1] ^ a[2])

a[2] ^= t ^ xtime(a[2] ^ a[3])

a[3] ^= t ^ xtime(a[3] ^ u)

def mix\_columns(state):

for i in range(4):

mix\_single\_column(state[i])

def inv\_mix\_columns(state):

for i in range(4):

u = xtime(xtime(state[i][0] ^ state[i][2]))

v = xtime(xtime(state[i][1] ^ state[i][3]))

state[i][0] ^= u

state[i][1] ^= v

state[i][2] ^= u

state[i][3] ^= v

def xtime(a):

return (((a << 1) ^ 0x1B) & 0xFF) if (a & 0x80) else (a << 1)

def key\_expansion(key):

key\_columns = bytes\_to\_matrix(key)

iteration\_size = len(key) // 4

columns = deepcopy(key\_columns)

i = 1

while len(columns) < 44:

word = list(columns[-1])

if len(columns) % iteration\_size == 0:

word.append(word.pop(0))

word = [s\_box[b] for b in word]

word[0] ^= r\_con[i % len(r\_con)]

i += 1

word = [a ^ b for a, b in zip(word, columns[-iteration\_size])]

columns.append(word)

round\_keys = [columns[4\*i:4\*(i+1)] for i in range(len(columns) // 4)]

return round\_keys

def encrypt\_block(plaintext, key):

state = bytes\_to\_matrix(plaintext)

round\_keys = key\_expansion(key)

add\_round\_key(state, round\_keys[0])

for r in range(1, len(round\_keys) - 1):

sub\_bytes(state)

shift\_rows(state)

mix\_columns(state)

add\_round\_key(state, round\_keys[r])

sub\_bytes(state)

shift\_rows(state)

add\_round\_key(state, round\_keys[-1])

return matrix\_to\_bytes(state)

def decrypt\_block(ciphertext, key):

state = bytes\_to\_matrix(ciphertext)

round\_keys = key\_expansion(key)

add\_round\_key(state, round\_keys[-1])

inv\_shift\_rows(state)

inv\_sub\_bytes(state)

for r in range(len(round\_keys) - 2, 0, -1):

add\_round\_key(state, round\_keys[r])

inv\_mix\_columns(state)

inv\_shift\_rows(state)

inv\_sub\_bytes(state)

add\_round\_key(state, round\_keys[0])

return matrix\_to\_bytes(state)

def hex\_to\_bytes(hex\_str):

return bytes.fromhex(hex\_str)

def string\_to\_bytes(text):

return bytes(text, 'utf-8')

def bytes\_to\_string(byte\_data):

return byte\_data.decode('utf-8')

def input\_text():

plaintext = input("Enter the plaintext (normal text): ")

key = input("Enter the key (16 characters, normal text): ")

plaintext\_bytes = string\_to\_bytes(plaintext)

key\_bytes = string\_to\_bytes(key)

if len(key\_bytes) != 16:

print("The key must be exactly 16 bytes long.")

return None, None

if len(plaintext\_bytes) < 1 or len(plaintext\_bytes) > 16 \* 16:

print("The plaintext must be between 1 byte and 16n bytes long.")

return None, None

return plaintext\_bytes, key\_bytes

def main():

plaintext, key = input\_text()

if plaintext and key:

print("Encrypting the plaintext...")

encrypted = encrypt\_block(plaintext, key)

print(f"Encrypted ciphertext (in hex): {encrypted.hex()}")

print("Decrypting the ciphertext...")

decrypted = decrypt\_block(encrypted, key)

print(f"Decrypted plaintext (normal text): {bytes\_to\_string(decrypted)}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

# **EXPLANATION**

AES (Advanced Encryption Standard) is a **symmetric block cipher** that encrypts **128-bit** blocks of data using a **128-bit key**. It operates in rounds, and AES-128 has **10 rounds** of transformation.

Each round consists of:

1. **SubBytes** – Byte substitution using an S-box.
2. **ShiftRows** – Row-wise permutation.
3. **MixColumns** – Column transformation (except in the last round).
4. **AddRoundKey** – XOR with a round key.

The provided Python code implements the fundamental components of the Advanced Encryption Standard (AES) algorithm. It defines the substitution box (S-box) and its inverse (inverse S-box) for the SubBytes and InvSubBytes transformations, along with the round constants (r\_con) utilized during the key expansion process. Utility functions such as bytes\_to\_matrix and matrix\_to\_bytes are included to facilitate the conversion between byte arrays and 4×4 matrices, which is essential for AES operations. Core cryptographic transformations are implemented: add\_round\_key executes a bitwise XOR between the state and the round key; sub\_bytes and inv\_sub\_bytes perform byte substitution using the respective S-boxes; and shift\_rows and inv\_shift\_rows introduce diffusion by cyclically shifting matrix rows. The mix\_single\_column and mix\_columns functions execute the MixColumns transformation to further enhance diffusion, while inv\_mix\_columns implements its inverse by computing intermediate field operations. The xtime function provides multiplication by 2 in the Galois Field GF(2^8), a critical operation within MixColumns. Furthermore, the key\_expansion function generates the necessary round keys by expanding the initial cipher key through a combination of byte substitution, word rotation, and XOR with the round constants. Collectively, this code establishes the core building blocks required for the encryption and decryption processes in AES.

**OUTPUT**



