```
#########################################################

####INTRO & PREPARATION####
#########################################################

#AES is a block cipher which encrypts plaintext in blocks
of 128-bit, using a key of 128, 192, or 256-bit. For the
illustration of the solution we will assume plaintext and
key to be hexadecimal, and the plaintext and key size to
be 128-bit. You are free and encouraged to use any other
encoding of your choice. Note that AES block, also called
state, can be represented as a four by four matrix of
bytes.

#Plaintext message as four by four matrix, with the
following text: "AES is quite fun"
m_four_by_four_matrix = [
[0x41, 0x69, 0x75, 0x20],
[0x45, 0x73, 0x69, 0x66],
[0x53, 0x20, 0x74, 0x75],
[0x20, 0x71, 0x65, 0x6e]
]

#Plaintext message as four by four matrix. You can use
this URL, https://www.kavaliro.com/wp-
content/uploads/2014/03/AES.pdf, and example, to follow
the steps for verifying that your impelementation is
working correctly.
#m_four_by_four_matrix = [
#[0x54,0x4F,0x4E,0x20],
#[0x77,0x6E,0x69,0x54],
#[0x6F,0x65,0x6E,0x77],
#[0x20,0x20,0x65,0x6F]
#]
```

```
#The key used in the example is the following:
#key =
'010101000110100001100001011101000111001100100000011011010

####Supportive function####
#Implement the encoding function of your choice. Here in
the example solution you find a function that uses
builtin functions bin() and int() to convert hexadecimal
to binary.
def hex_to_binary(hex_digits):
        binary_digits = ""
        for hex_digit in hex_digits:
                binary_digits += bin(int(hex_digit,16))
[2:].zfill(4)
        return binary_digits


##########################################################

####APPLYING S-BOXES####
##########################################################

#The AES s-boxes
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]]

#The AES inverted s-boxes
invers_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]]

###TODO###
#Implement a functions that returns the row and coloumn
index from the s-box based on the 8-bit (one byte) input.
def get_sbox_indexes(key_byte):
    #// Enter you code here

###TODO###
#Implement a function that given the row and column index
of the s-box returns the corresponding entry.
def get_sbox_entry(row_index, column_index):
    #// Enter you code here

###TODO###
#Implement a function that applies the s-box to the AES
state, i.e. writes s-box corresponding values into a four
```

```
by four matrix. Note that this function also implements
the byte substitution layer.
def byte_substitution(fbf_matrix):
    #// Enter you code here



##########################################################


####KEY DERIVATION####
##########################################################


###TODO###
#Generate a pseudo random "secret" key of 128-bit.
Suggested to use secrets.token_hex, and supportive
function hex_to_binary.
    #// Enter you code here

###TODO###
#Implement a function that performes a left shift of a
word, i.e. of a 32-bit.
def byte_left_shift(word_from_key):
    #// Enter you code here

###TODO###
#Implement the g-Function of AES. Note that the g-
Function, uses the round coefficients in dependence of
the current round. Therefore, parametrise the g-Function
such that the current round can be entered as a
parameter.
def g_Function(word_from_key, round):
    #// Enter you code here

#Round Coefficients
round_coefficient =
```

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

###TODO###
#Implement a function that generates and returns all
round keys. Note that the number of round keys depends on
the key size. For simplicity we assume a key size of 128-
bit, i.e. 10 rounds.
def generate_round_keys(secret_key):
    #// Enter you code here



##########################################################


####SHIFTROW LAYER####
##########################################################


###TODO###
#Implement a function that performs a bytwise left
ciruclar shift on the state matrix, i.e. on a four by
four matrix.
def shift_rows(fbf_matrix):
    #// Enter you code here



##########################################################


####MIXCOLUMN LAYER####
##########################################################


#The matrix used for the linear transformation of each
column of the state matrix.
```

```
mix_column_matrix=[
[0x02,0x03,0x01,0x01],
[0x01,0x02,0x03,0x01],
[0x01,0x01,0x02,0x03],
[0x03,0x01,0x01,0x02]]

####Supportive function####
#Implements the Galois Field multiplication.
def gf_multiplication(a, b):
    if b == 1:
        return a
    tmp = (a << 1) & 0xff
    if b == 2:
        return tmp if a < 128 else tmp ^ 0x1b
    if b == 3:
        return gf_multiplication(a, 2) ^ a

###TODO###
#Implement a function that multiplies each column of the
state matrix with the given matrix.
def mix_columns(mc_matrix,fbf_matrix):
    #// Enter you code here



##########################################################



####ADD ROUND KEY LAYER####
##########################################################


###TODO###
#Implement a function that takes a 128-bit string and
stores it in a 4x4 matrix.
def bin_key_to_matrix(bin_key):
    #// Enter you code here
```

```
###TODO###
#Implement a function that performs the XOR operation
between the AES state and the key
def add_round_key(fbf_matrix, k_matrix):
    #// Enter you code here



###########################################################


####AES ENCRYPTION####
###########################################################


####Supportive function####
#Implements a function that returns a matrix in
hexadecimal.
def matrix_to_hex(fbf_matrix):
    #// Enter you code here

###TODO###
#Implement the AES encryption function with the help of
the functions that implemented above. Assume a 128-key
bit, i.e. 10 rounds.
def encrypt(message, key):
    #// Enter you code here
```