AES in python

Here we'll be implementing AES algorithm in python

Submitted by: *Shivakumar B* **Roll no:** *CS22B1052*

We'll be taking a random plain text, a key and the expected output in hex We'll encode them in hex

```
import codecs
plaintext="6bc1bee22e409f96e93d7e117393172a"
expected_output="3ad77bb40d7a3660a89ecaf32466ef97"
key="2b7e151628aed2a6abf7158809cf4f3c"
keys=codecs.decode(key,'hex')


# for i in range(0,len(key),2):
#     keys.append(int(key[i:i+2],16))
# print(type(key))
# print(keys)
```

RCon and SBox *Here we have constructed the RCon and the SBox We have also made an SBox to SboxInv test to show how the substitution box works*

```
Rcon = ( 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b,
0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a )
print(type(Rcon[0]))
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
        )
Sbox_inv = (
            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
            )
```

```
print(Sbox_inv[Sbox[0x52]])
print(keys[15])

<class 'int'>
82
60
```

Key expansion *Here we have applied the key expansion algorithm specified*

```
def xor(s1, s2):
    return tuple(a^b for a,b in zip(s1, s2))

def rot_word(word):
    return word[1:]+word[:1]

def sub_word(word):
    # for w in word:
        # print(type(w))
    return (Sbox[w] for w in word)

def key_expansion(keys):
    word=[]
    for i in range(4):

word.append(tuple([keys[4*i],keys[4*i+1],keys[4*i+2],keys[4*i+3]]))

    for i in range(4,44):
        temp=word[i-1]
        if i%4==0:
            temp=xor(sub_word(rot_word(temp)),(Rcon[i//4],0,0,0))
        # print(i,word[i])
        word.append(xor(word[i-4],temp))

    return word

keyl=key_expansion(keys)
print(len(keyl))

44
```

Multiplication table *Here we are storing all the possible pairs for multiplication*

```
state=map(ord,plaintext.encode().hex())
Gmul = {}
def gmul(a, b):
        p = 0
        for c in range(8):
            if b & 1:
                p ^= a
            a <<= 1
            if a & 0x100:
                a ^= 0x11b
            b >>= 1
        return p
for f in (0x02, 0x03, 0x0e, 0x0b, 0x0d, 0x09):
    Gmul[f] = tuple(gmul(f, x) for x in range(0,0x100))

state=list(state)
```

Substitute bytes *Implementation of the first transformation function i.e., Substitute bytes*

```
def sub_bytes(state):
    # print(state)
    for i,b in enumerate(state):
        state[i]=Sbox[b]
    return state

def inv_sub_bytes(state):
    for i,b in enumerate(state):
        state[i]=Sbox_inv[b]
    return state
```

Shift rows *Implementation of the second transformation function i.e., Shift rows*

```
def shift_rows(state):
    rows = []
    for r in range(4):
        rows.append( state[r::4] )
        rows[r] = rows[r][r:] + rows[r][:r]
    state = [ r[c] for c in range(4) for r in rows ]
    return state
state=sub_bytes(state)
state=shift_rows(state)
print(state)

def inv_shift_rows(state):
    rows = []
    for r in range(4):
        rows.append( state[r::4] )
        rows[r] = rows[r][4-r:] + rows[r][:4-r]
    state = [ r[c] for c in range(4) for r in rows ]
    return state
```

```
[195, 195, 5, 35, 5, 35, 195, 150, 5, 150, 5, 4, 5, 35, 195, 5]
```

Substitute bytes *Implementation of the third transformation function i.e., Adding the Round key*

```
def add_round_key(state, rkey):
    # print(len(state))
    # print(len(rkey))
    for i, b in enumerate(rkey):
        # state[i] ^= b
        for l,j in enumerate(b):
            state[i*4+l]^=j

    return state
```

Mix Columns *Implementation of the fourth and final transformation function i.e., Mix columns*

```
def mix_columns(state):
    ss = []
    for c in range(4):
        col = state[c*4:(c+1)*4]
        ss.extend((
                    Gmul[0x02][col[0]] ^ Gmul[0x03][col[1]] ^
col[2]  ^                col[3] ,
                                col[0]  ^ Gmul[0x02][col[1]] ^
Gmul[0x03][col[2]] ^            col[3] ,
                                col[0]  ^                col[1]  ^
Gmul[0x02][col[2]] ^ Gmul[0x03][col[3]],
                    Gmul[0x03][col[0]] ^                col[1]  ^
col[2]  ^ Gmul[0x02][col[3]],
                ))
    return ss

state=mix_columns(state)
state

def inv_mix_columns(state):
    ss = []
    for c in range(4):
        col = state[c*4:(c+1)*4]
        ss.extend((
                    Gmul[0x0e][col[0]] ^ Gmul[0x0b][col[1]] ^
Gmul[0x0d][col[2]] ^ Gmul[0x09][col[3]],
                    Gmul[0x09][col[0]] ^ Gmul[0x0e][col[1]] ^
Gmul[0x0b][col[2]] ^ Gmul[0x0d][col[3]],
                    Gmul[0x0d][col[0]] ^ Gmul[0x09][col[1]] ^
Gmul[0x0e][col[2]] ^ Gmul[0x0b][col[3]],
                    Gmul[0x0b][col[0]] ^ Gmul[0x0d][col[1]] ^
Gmul[0x09][col[2]] ^ Gmul[0x0e][col[3]],
                ))
    return ss
```

Entire cipher operation

```python
def cipher(pt,key):
    state=codecs.decode(pt,'hex')
    state=list(state)
    keys=key_expansion(key)
    # print(keys)
    add_round_key(state,keys[0:4])
    for i in range(1,10):
        state=sub_bytes(state)

        state=shift_rows(state)

        state=mix_columns(state)

        k=keys[4*i:4*(i+1)]

        state=add_round_key(state,k)

        curr="".join(map(chr,state))
        print(f"after round {i}: {curr}")

    state=sub_bytes(state)
    state=shift_rows(state)
    state=add_round_key(state,keys[40:])

    return "".join(map(chr,state))


ct=cipher(plaintext,keys)

print(f"\nfinal result:")
ctinhex=""
for i in range(len(ct)):
    ctinhex+=f"{format(ord(ct[i]),"x")}"
print(f"\nIn hex:")
print(ctinhex)
print(f"\nIn character string:")
print(ct)

hello=list(codecs.decode("3ad77bb40d7a3660a89ecaf32466ef97",'hex'))
hello="".join(map(chr,hello))
print(f"\nExpected result for this input:")
print(f"\nIn hex:")
print(expected_output)
print(f"\nIn character string:")
print(hello)
```

```
after round 1: òeèÕÒ9{Ã¹□m□vP\
after round 2: ýó|ÛKł÷üØéJ©»ø
after round 3: ¬Ñì□¢BâÃiz·¹
after round 4: ¢an_D¥M9À)â□·dé
after round 5: ,Jó2ÃïÉÈ©¸{%.Í§
after round 6: ÍMÀ~³º□¹9ÿ+Ó¼÷
after round 7: âm»}@Ò!4ã·ý¢k□|
after round 8: A×ÆS}f□@Ý/□¬Å
after round 9: »6Çë❪3MI¤ç.tñ❨Ä

final result:

In hex:
3ad77bb4d7a3660a89ecaf32466ef97

In character string:
z6`¨□Êó$fï□

Expected result for this input:

In hex:
3ad77bb40d7a3660a89ecaf32466ef97

In character string:
z6`¨□Êó$fï□

def inv_cipher(ct,key):
    state=codecs.decode(ct,'hex')
    state=list(state)
    keys=key_expansion(key)
    add_round_key(state,keys[4*10:])

    for i in range(9,0,-1):
        state=inv_shift_rows(state)

        state=inv_sub_bytes(state)

        k=keys[i*4:(i+1)*4]

        state=add_round_key(state,k)

        state=inv_mix_columns(state)

        curr="".join(map(chr,state))
        # print(f"after round {i}: {curr}")

    state=inv_shift_rows(state)
    state=inv_sub_bytes(state)
    state=add_round_key(state,keys[0:4])
    return "".join(map(chr,state))

pt_dec=inv_cipher(expected_output,keys)
pt_decinhex=""
for i in range(len(pt_dec)):
    pt_decinhex+=f"{format(ord(pt_dec[i]),"x")}"
```

```python
print(f"original plain text: {plaintext}\n")
print(f"cipher text: {ctinhex}\n")
print(f"deciphered cipher text: {pt_decinhex}\n")

# print(ansinhex)
```

```
original plain text: 6bc1bee22e409f96e93d7e117393172a

cipher text: 3ad77bb4d7a3660a89ecaf32466ef97

deciphered cipher text: 6bc1bee22e409f96e93d7e117393172a
```