Name:-Amrit Ranjan

Reg No:-21BCE1847

Cryptography And Network Security

Assignment 1

Faculty Name:-Dr Tapabarta Roy



Aes Encryption

Python Code:-

s\_box\_string = '63 7c 77 7b f2 6b 6f c5 30 01 67 2b fe d7 ab 76' \

'ca 82 c9 7d fa 59 47 f0 ad d4 a2 af 9c a4 72 c0' \

'b7 fd 93 26 36 3f f7 cc 34 a5 e5 f1 71 d8 31 15' \

'04 c7 23 c3 18 96 05 9a 07 12 80 e2 eb 27 b2 75' \

'09 83 2c 1a 1b 6e 5a a0 52 3b d6 b3 29 e3 2f 84' \

'53 d1 00 ed 20 fc b1 5b 6a cb be 39 4a 4c 58 cf' \

'd0 ef aa fb 43 4d 33 85 45 f9 02 7f 50 3c 9f a8' \

'51 a3 40 8f 92 9d 38 f5 bc b6 da 21 10 ff f3 d2' \

'cd 0c 13 ec 5f 97 44 17 c4 a7 7e 3d 64 5d 19 73' \

'60 81 4f dc 22 2a 90 88 46 ee b8 14 de 5e 0b db' \

'e0 32 3a 0a 49 06 24 5c c2 d3 ac 62 91 95 e4 79' \

'e7 c8 37 6d 8d d5 4e a9 6c 56 f4 ea 65 7a ae 08' \

'ba 78 25 2e 1c a6 b4 c6 e8 dd 74 1f 4b bd 8b 8a' \

'70 3e b5 66 48 03 f6 0e 61 35 57 b9 86 c1 1d 9e' \

'e1 f8 98 11 69 d9 8e 94 9b 1e 87 e9 ce 55 28 df' \

'8c a1 89 0d bf e6 42 68 41 99 2d 0f b0 54 bb 16'.replace(" ", "")

s\_box = bytearray.fromhex(s\_box\_string)

def sub\_word(word: [int]) -> bytes:

substituted\_word = bytes(s\_box[i] for i in word)

return substituted\_word

def rcon(i: int) -> bytes:

# From Wikipedia

rcon\_lookup = bytearray.fromhex('01020408102040801b36')

rcon\_value = bytes([rcon\_lookup[i-1], 0, 0, 0])

return rcon\_value

def xor\_bytes(a: bytes, b: bytes) -> bytes:

return bytes([x ^ y for (x, y) in zip(a, b)])

def rot\_word(word: [int]) -> [int]:

return word[1:] + word[:1]

def key\_expansion(key: bytes, nb: int = 4) -> [[[int]]]:

nk = len(key) // 4

key\_bit\_length = len(key) \* 8

if key\_bit\_length == 128:

nr = 10

elif key\_bit\_length == 192:

nr = 12

else: # 256-bit keys

nr = 14

w = state\_from\_bytes(key)

for i in range(nk, nb \* (nr + 1)):

temp = w[i-1]

if i % nk == 0:

temp = xor\_bytes(sub\_word(rot\_word(temp)), rcon(i // nk))

elif nk > 6 and i % nk == 4:

temp = sub\_word(temp)

w.append(xor\_bytes(w[i - nk], temp))

return [w[i\*4:(i+1)\*4] for i in range(len(w) // 4)]

def add\_round\_key(state: [[int]], key\_schedule: [[[int]]], round: int):

round\_key = key\_schedule[round]

for r in range(len(state)):

state[r] = [state[r][c] ^ round\_key[r][c] for c in range(len(state[0]))]

def sub\_bytes(state: [[int]]):

for r in range(len(state)):

state[r] = [s\_box[state[r][c]] for c in range(len(state[0]))]

def shift\_rows(state: [[int]]):

# [00, 10, 20, 30] [00, 10, 20, 30]

# [01, 11, 21, 31] --> [11, 21, 31, 01]

# [02, 12, 22, 32] [22, 32, 02, 12]

# [03, 13, 23, 33] [33, 03, 13, 23]

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

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

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

def xtime(a: int) -> int:

if a & 0x80:

return ((a << 1) ^ 0x1b) & 0xff

return a << 1

def mix\_column(col: [int]):

c\_0 = col[0]

all\_xor = col[0] ^ col[1] ^ col[2] ^ col[3]

col[0] ^= all\_xor ^ xtime(col[0] ^ col[1])

col[1] ^= all\_xor ^ xtime(col[1] ^ col[2])

col[2] ^= all\_xor ^ xtime(col[2] ^ col[3])

col[3] ^= all\_xor ^ xtime(c\_0 ^ col[3])

def mix\_columns(state: [[int]]):

for r in state:

mix\_column(r)

def state\_from\_bytes(data: bytes) -> [[int]]:

state = [data[i\*4:(i+1)\*4] for i in range(len(data) // 4)]

return state

def bytes\_from\_state(state: [[int]]) -> bytes:

return bytes(state[0] + state[1] + state[2] + state[3])

def aes\_encryption(data: bytes, key: bytes) -> bytes:

key\_bit\_length = len(key) \* 8

if key\_bit\_length == 128:

nr = 10

elif key\_bit\_length == 192:

nr = 12

else: # 256-bit keys

nr = 14

state = state\_from\_bytes(data)

key\_schedule = key\_expansion(key)

add\_round\_key(state, key\_schedule, round=0)

for round in range(1, nr):

sub\_bytes(state)

shift\_rows(state)

mix\_columns(state)

add\_round\_key(state, key\_schedule, round)

sub\_bytes(state)

shift\_rows(state)

add\_round\_key(state, key\_schedule, round=nr)

cipher = bytes\_from\_state(state)

return cipher

def inv\_shift\_rows(state: [[int]]) -> [[int]]:

# [00, 10, 20, 30] [00, 10, 20, 30]

# [01, 11, 21, 31] <-- [11, 21, 31, 01]

# [02, 12, 22, 32] [22, 32, 02, 12]

# [03, 13, 23, 33] [33, 03, 13, 23]

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

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

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

return

inv\_s\_box\_string = '52 09 6a d5 30 36 a5 38 bf 40 a3 9e 81 f3 d7 fb' \

'7c e3 39 82 9b 2f ff 87 34 8e 43 44 c4 de e9 cb' \

'54 7b 94 32 a6 c2 23 3d ee 4c 95 0b 42 fa c3 4e' \

'08 2e a1 66 28 d9 24 b2 76 5b a2 49 6d 8b d1 25' \

'72 f8 f6 64 86 68 98 16 d4 a4 5c cc 5d 65 b6 92' \

'6c 70 48 50 fd ed b9 da 5e 15 46 57 a7 8d 9d 84' \

'90 d8 ab 00 8c bc d3 0a f7 e4 58 05 b8 b3 45 06' \

'd0 2c 1e 8f ca 3f 0f 02 c1 af bd 03 01 13 8a 6b' \

'3a 91 11 41 4f 67 dc ea 97 f2 cf ce f0 b4 e6 73' \

'96 ac 74 22 e7 ad 35 85 e2 f9 37 e8 1c 75 df 6e' \

'47 f1 1a 71 1d 29 c5 89 6f b7 62 0e aa 18 be 1b' \

'fc 56 3e 4b c6 d2 79 20 9a db c0 fe 78 cd 5a f4' \

'1f dd a8 33 88 07 c7 31 b1 12 10 59 27 80 ec 5f' \

'60 51 7f a9 19 b5 4a 0d 2d e5 7a 9f 93 c9 9c ef' \

'a0 e0 3b 4d ae 2a f5 b0 c8 eb bb 3c 83 53 99 61' \

'17 2b 04 7e ba 77 d6 26 e1 69 14 63 55 21 0c 7d'.replace(" ", "")

inv\_s\_box = bytearray.fromhex(inv\_s\_box\_string)

def inv\_sub\_bytes(state: [[int]]) -> [[int]]:

for r in range(len(state)):

state[r] = [inv\_s\_box[state[r][c]] for c in range(len(state[0]))]

def xtimes\_0e(b):

# 0x0e = 14 = b1110 = ((x \* 2 + x) \* 2 + x) \* 2

return xtime(xtime(xtime(b) ^ b) ^ b)

def xtimes\_0b(b):

# 0x0b = 11 = b1011 = ((x\*2)\*2+x)\*2+x

return xtime(xtime(xtime(b)) ^ b) ^ b

def xtimes\_0d(b):

# 0x0d = 13 = b1101 = ((x\*2+x)\*2)\*2+x

return xtime(xtime(xtime(b) ^ b)) ^ b

def xtimes\_09(b):

# 0x09 = 9 = b1001 = ((x\*2)\*2)\*2+x

return xtime(xtime(xtime(b))) ^ b

def inv\_mix\_column(col: [int]):

c\_0, c\_1, c\_2, c\_3 = col[0], col[1], col[2], col[3]

col[0] = xtimes\_0e(c\_0) ^ xtimes\_0b(c\_1) ^ xtimes\_0d(c\_2) ^ xtimes\_09(c\_3)

col[1] = xtimes\_09(c\_0) ^ xtimes\_0e(c\_1) ^ xtimes\_0b(c\_2) ^ xtimes\_0d(c\_3)

col[2] = xtimes\_0d(c\_0) ^ xtimes\_09(c\_1) ^ xtimes\_0e(c\_2) ^ xtimes\_0b(c\_3)

col[3] = xtimes\_0b(c\_0) ^ xtimes\_0d(c\_1) ^ xtimes\_09(c\_2) ^ xtimes\_0e(c\_3)

def inv\_mix\_columns(state: [[int]]) -> [[int]]:

for r in state:

inv\_mix\_column(r)

def inv\_mix\_column\_optimized(col: [int]):

u = xtime(xtime(col[0] ^ col[2]))

v = xtime(xtime(col[1] ^ col[3]))

col[0] ^= u

col[1] ^= v

col[2] ^= u

col[3] ^= v

def inv\_mix\_columns\_optimized(state: [[int]]) -> [[int]]:

for r in state:

inv\_mix\_column\_optimized(r)

mix\_columns(state)

def aes\_decryption(cipher: bytes, key: bytes) -> bytes:

key\_byte\_length = len(key)

key\_bit\_length = key\_byte\_length \* 8

nk = key\_byte\_length // 4

if key\_bit\_length == 128:

nr = 10

elif key\_bit\_length == 192:

nr = 12

else: # 256-bit keys

nr = 14

state = state\_from\_bytes(cipher)

key\_schedule = key\_expansion(key)

add\_round\_key(state, key\_schedule, round=nr)

for round in range(nr-1, 0, -1):

inv\_shift\_rows(state)

inv\_sub\_bytes(state)

add\_round\_key(state, key\_schedule, round)

inv\_mix\_columns(state)

inv\_shift\_rows(state)

inv\_sub\_bytes(state)

add\_round\_key(state, key\_schedule, round=0)

plain = bytes\_from\_state(state)

return plain

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

# NIST FIPS PUB 197 ADVANCED ENCRYPTION STANDARD (AES)

# NIST AES-128 test vector 1 (Ch. C.1, p. 35)

plaintext = bytearray.fromhex('00112233445566778899aabbccddeeff')

key = bytearray.fromhex('000102030405060708090a0b0c0d0e0f')

expected\_ciphertext = bytearray.fromhex('69c4e0d86a7b0430d8cdb78070b4c55a')

print("Original Plaintext:", plaintext.hex())

ciphertext = aes\_encryption(plaintext, key)

print("Ciphertext:", ciphertext.hex())

recovered\_plaintext = aes\_decryption(ciphertext, key)

assert (ciphertext == expected\_ciphertext)

print("Recovered Plaintext:", recovered\_plaintext.hex())

assert (recovered\_plaintext == plaintext)

print("AES-128 Test Passed!")

# NIST AES-192 test vector 2 (Ch. C.2, p. 38)

plaintext = bytearray.fromhex('00112233445566778899aabbccddeeff')

key = bytearray.fromhex('000102030405060708090a0b0c0d0e0f1011121314151617')

expected\_ciphertext = bytearray.fromhex('dda97ca4864cdfe06eaf70a0ec0d7191')

ciphertext = aes\_encryption(plaintext, key)

recovered\_plaintext = aes\_decryption(ciphertext, key)

assert (ciphertext == expected\_ciphertext)

assert (recovered\_plaintext == plaintext)

# NIST AES-256 test vector 3 (Ch. C.3, p. 42)

plaintext = bytearray.fromhex('00112233445566778899aabbccddeeff')

key = bytearray.fromhex('000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f')

expected\_ciphertext = bytearray.fromhex('8ea2b7ca516745bfeafc49904b496089')

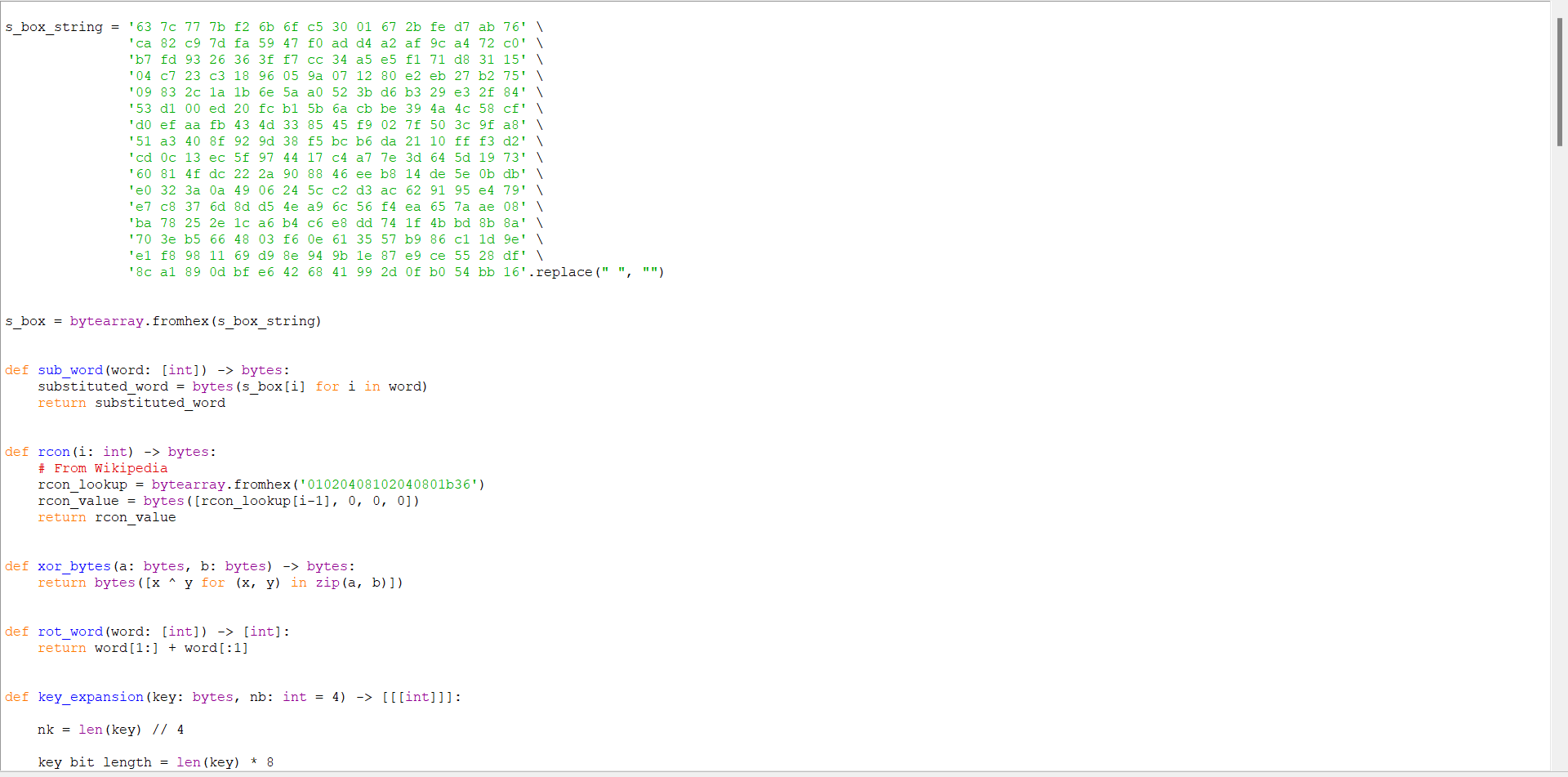
ciphertext = aes\_encryption(plaintext, key)

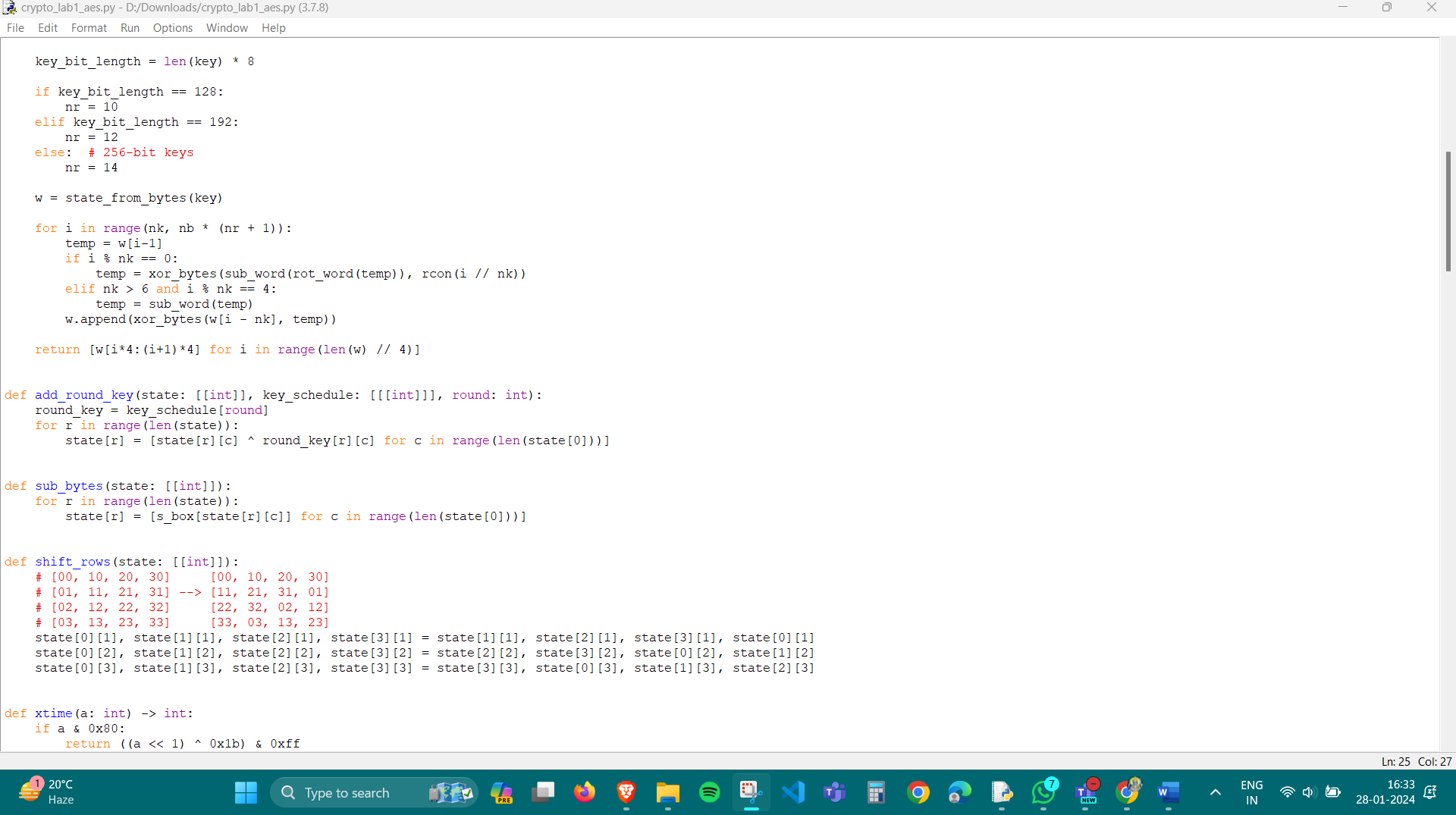
recovered\_plaintext = aes\_decryption(ciphertext, key)

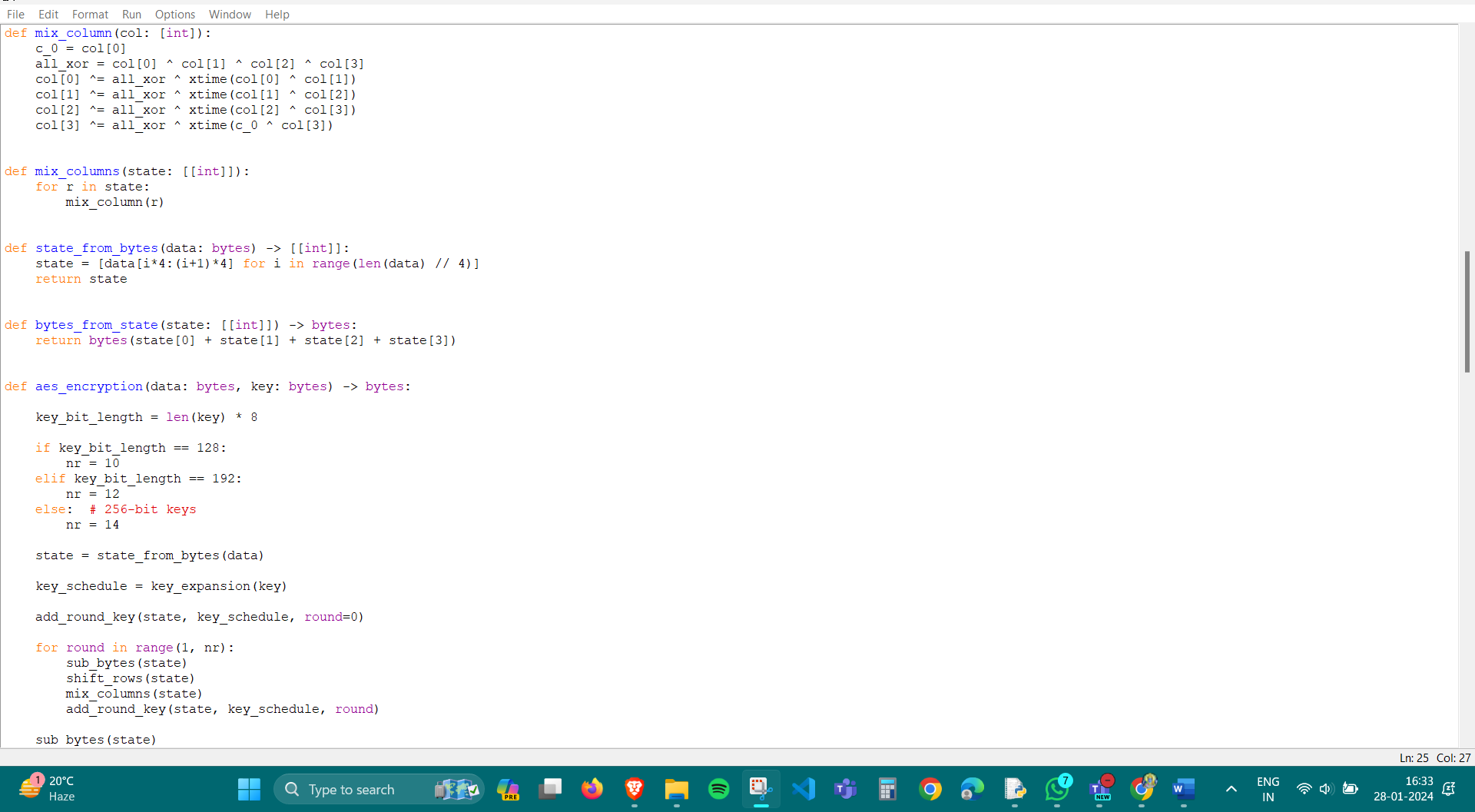
assert (ciphertext == expected\_ciphertext)

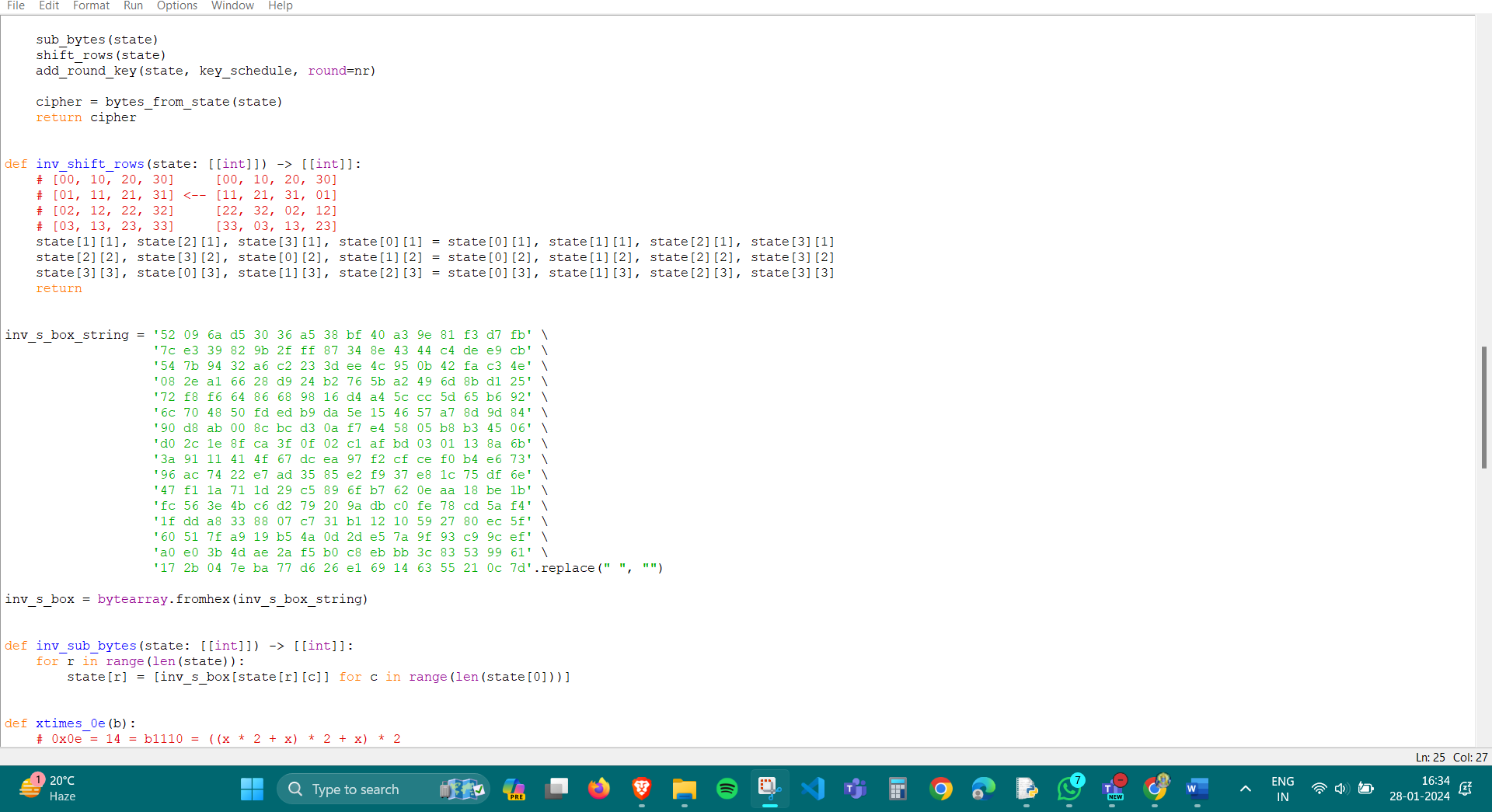
assert (recovered\_plaintext == plaintext)

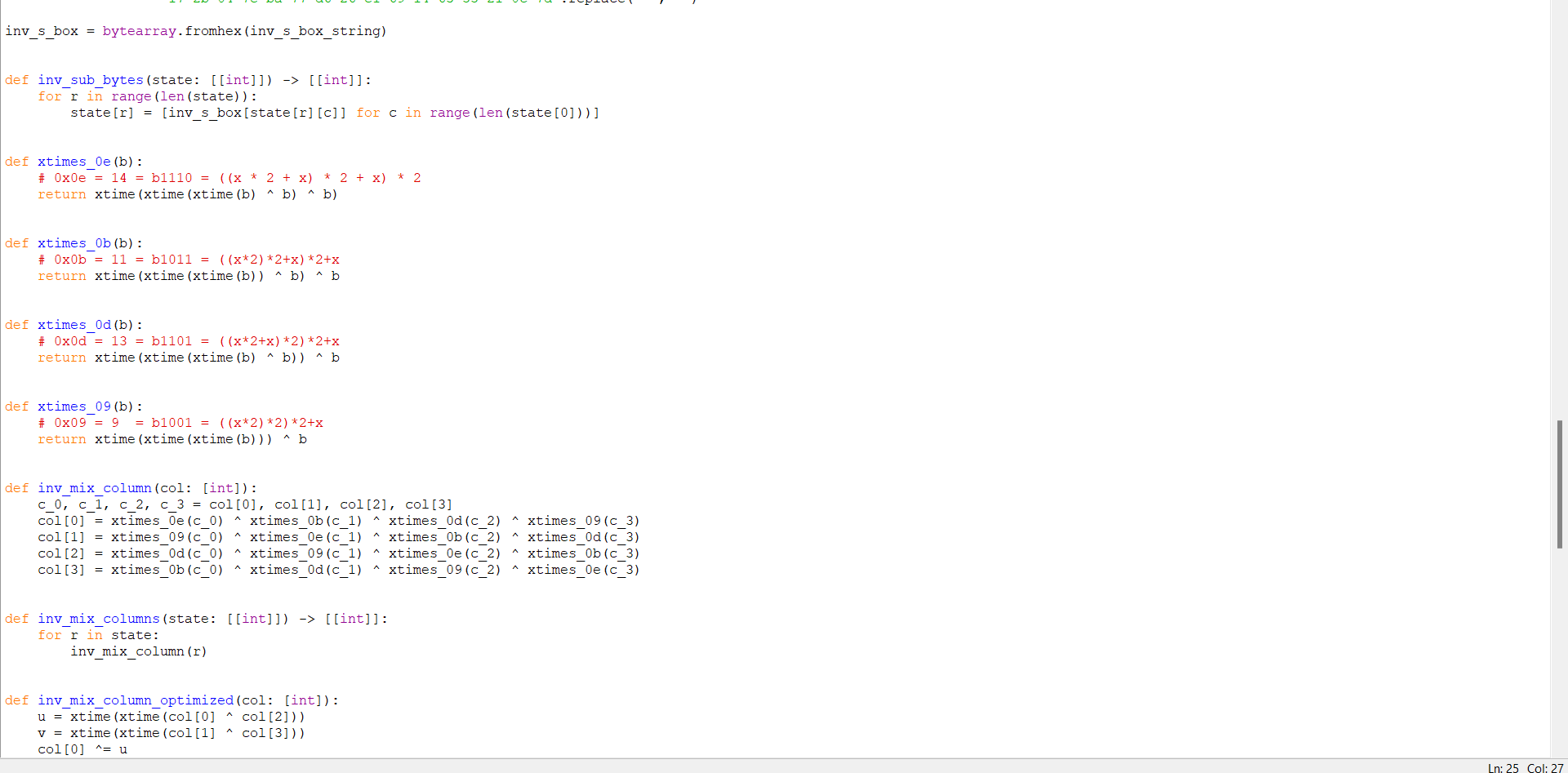
Screen Shots:-

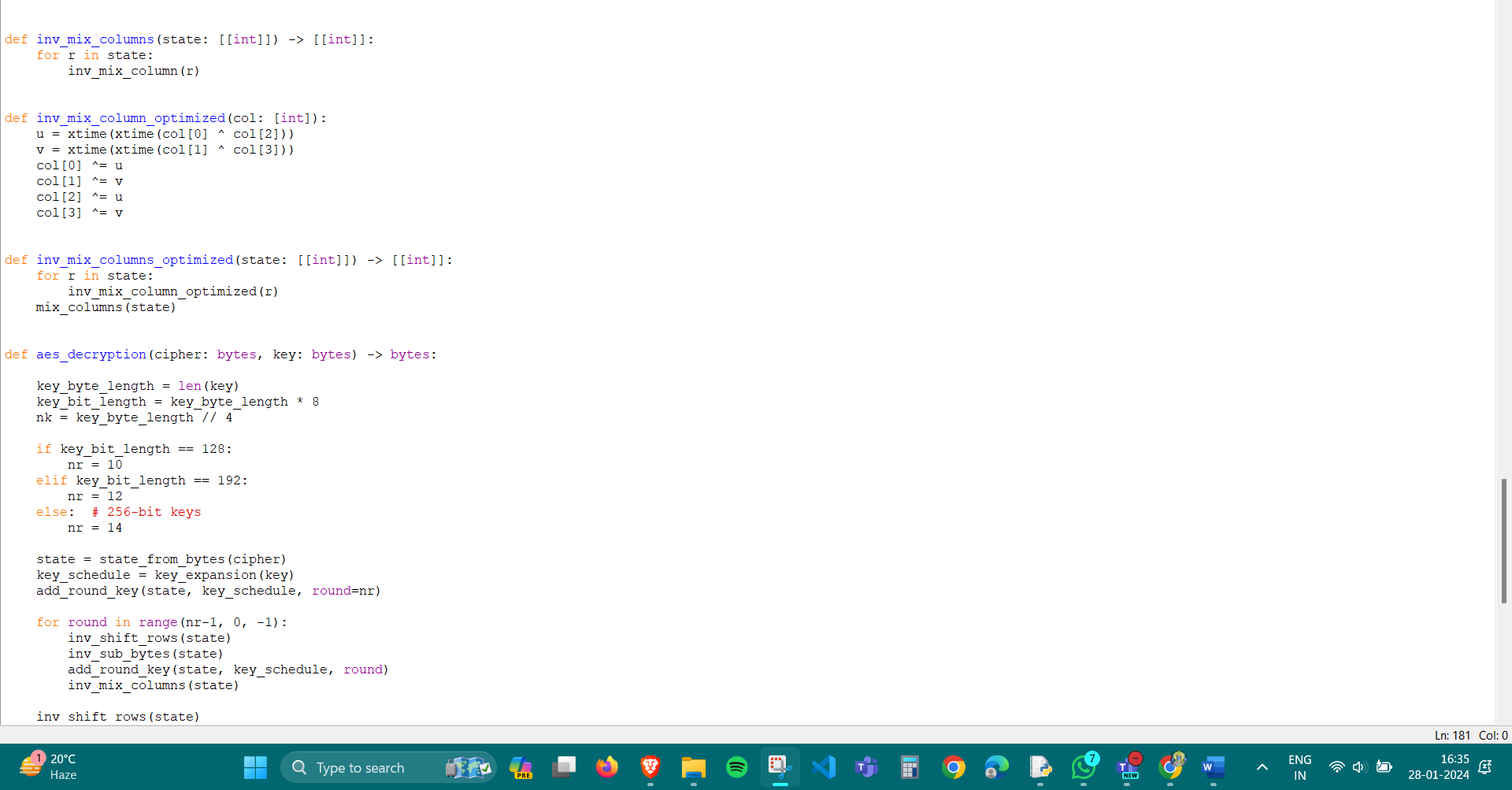


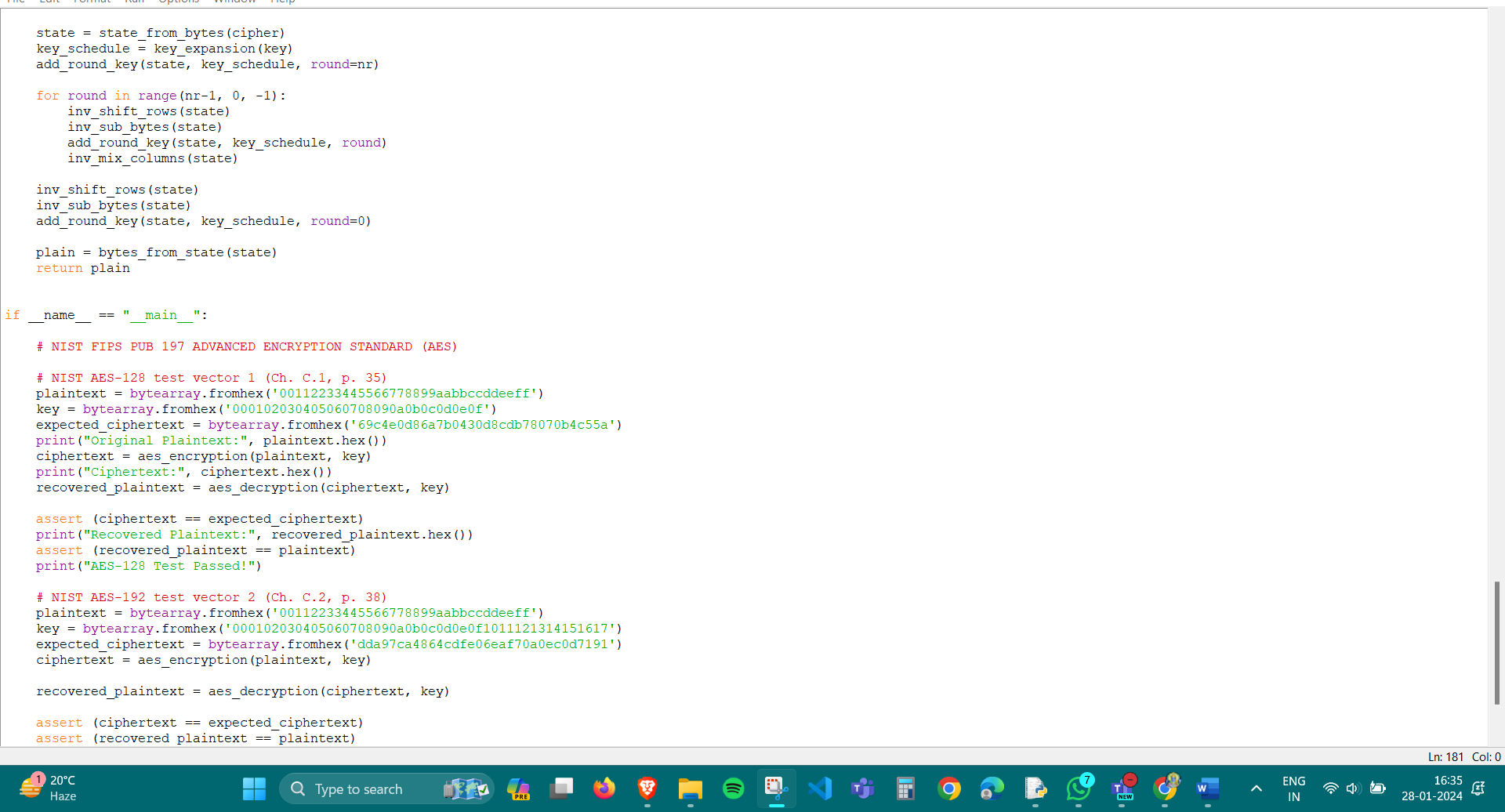


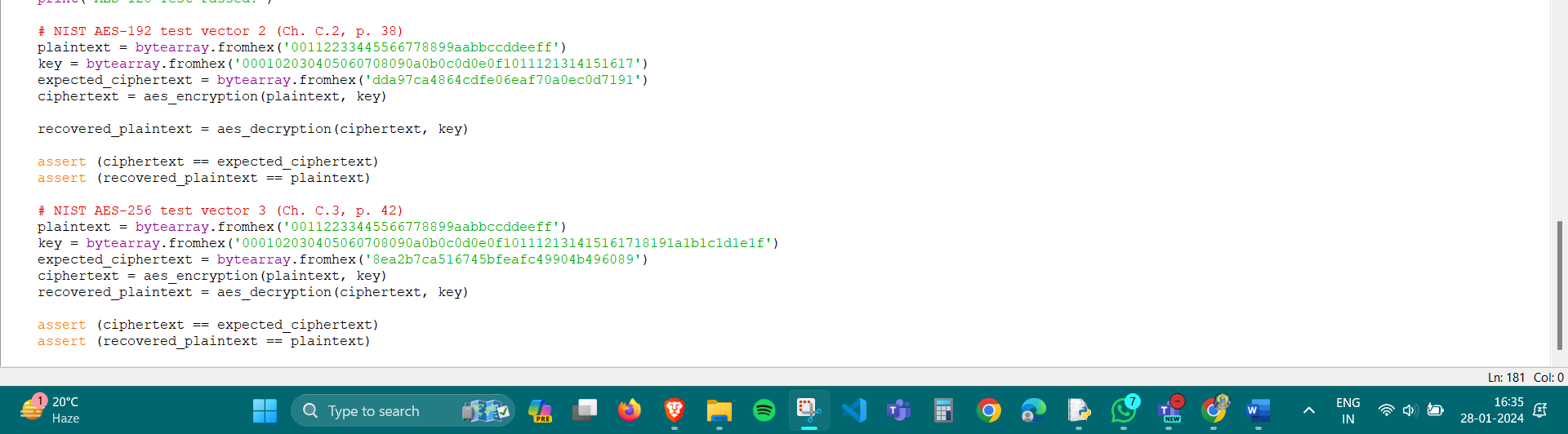












Output:-

