#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

"""Simple AES cipher implementation in pure Python following PEP-272 API

Based on: https://bitbucket.org/intgr/pyaes/ to compatible with PEP-8.

The goal of this module is to be as fast as reasonable in Python while still

being Pythonic and readable/understandable. It is licensed under the permissive

MIT license.

Hopefully the code is readable and commented enough that it can serve as an

introduction to the AES cipher for Python coders. In fact, it should go along

well with the Stick Figure Guide to AES:

http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html

Contrary to intuition, this implementation numbers the 4x4 matrices from top to

bottom for efficiency reasons::

0 4 8 12

1 5 9 13

2 6 10 14

3 7 11 15

Effectively it's the transposition of what you'd expect. This actually makes

the code simpler -- except the ShiftRows step, but hopefully the explanation

there clears it up.

"""

####

# Copyright (c) 2010 Marti Raudsepp <marti@juffo.org>

#

# Permission is hereby granted, free of charge, to any person obtaining a copy

# of this software and associated documentation files (the "Software"), to deal

# in the Software without restriction, including without limitation the rights

# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

# copies of the Software, and to permit persons to whom the Software is

# furnished to do so, subject to the following conditions:

#

# The above copyright notice and this permission notice shall be included in

# all copies or substantial portions of the Software.

#

# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN

# THE SOFTWARE.

####

from array import array

# Globals mandated by PEP 272:

# http://www.python.org/dev/peps/pep-0272/

MODE\_ECB = 1

MODE\_CBC = 2

#MODE\_CTR = 6

block\_size = 16

# variable length key: 16, 24 or 32 bytes

key\_size = None

class AESDecrypter():

MODE\_CBC=2

def new(self,key, mode, IV=None):

if mode == MODE\_ECB:

return ECBMode(AES(key))

elif mode == MODE\_CBC:

if IV is None:

raise ValueError("CBC mode needs an IV value!")

return CBCMode(AES(key), IV)

else:

raise NotImplementedError

#### AES cipher implementation

class AES(object):

block\_size = 16

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

self.setkey(key)

def setkey(self, key):

"""Sets the key and performs key expansion."""

self.key = key

self.key\_size = len(key)

if self.key\_size == 16:

self.rounds = 10

elif self.key\_size == 24:

self.rounds = 12

elif self.key\_size == 32:

self.rounds = 14

else:

raise ValueError("Key length must be 16, 24 or 32 bytes")

self.expand\_key()

def expand\_key(self):

"""Performs AES key expansion on self.key and stores in self.exkey"""

# The key schedule specifies how parts of the key are fed into the

# cipher's round functions. "Key expansion" means performing this

# schedule in advance. Almost all implementations do this.

#

# Here's a description of AES key schedule:

# http://en.wikipedia.org/wiki/Rijndael\_key\_schedule

# The expanded key starts with the actual key itself

exkey = array('B', self.key)

# extra key expansion steps

if self.key\_size == 16:

extra\_cnt = 0

elif self.key\_size == 24:

extra\_cnt = 2

else:

extra\_cnt = 3

# 4-byte temporary variable for key expansion

word = exkey[-4:]

# Each expansion cycle uses 'i' once for Rcon table lookup

for i in xrange(1, 11):

#### key schedule core:

# left-rotate by 1 byte

word = word[1:4] + word[0:1]

# apply S-box to all bytes

for j in xrange(4):

word[j] = aes\_sbox[word[j]]

# apply the Rcon table to the leftmost byte

word[0] ^= aes\_Rcon[i]

#### end key schedule core

for z in xrange(4):

for j in xrange(4):

# mix in bytes from the last subkey

word[j] ^= exkey[-self.key\_size + j]

exkey.extend(word)

# Last key expansion cycle always finishes here

if len(exkey) >= (self.rounds + 1) \* self.block\_size:

break

# Special substitution step for 256-bit key

if self.key\_size == 32:

for j in xrange(4):

# mix in bytes from the last subkey XORed with S-box of

# current word bytes

word[j] = aes\_sbox[word[j]] ^ exkey[-self.key\_size + j]

exkey.extend(word)

# Twice for 192-bit key, thrice for 256-bit key

for z in xrange(extra\_cnt):

for j in xrange(4):

# mix in bytes from the last subkey

word[j] ^= exkey[-self.key\_size + j]

exkey.extend(word)

self.exkey = exkey

def add\_round\_key(self, block, round):

"""AddRoundKey step. This is where the key is mixed into plaintext"""

offset = round \* 16

exkey = self.exkey

for i in xrange(16):

block[i] ^= exkey[offset + i]

#print 'AddRoundKey:', block

def sub\_bytes(self, block, sbox):

"""

SubBytes step, apply S-box to all bytes

Depending on whether encrypting or decrypting, a different sbox array

is passed in.

"""

for i in xrange(16):

block[i] = sbox[block[i]]

#print 'SubBytes :', block

def shift\_rows(self, b):

"""

ShiftRows step in AES.

Shifts 2nd row to left by 1, 3rd row by 2, 4th row by 3

Since we're performing this on a transposed matrix, cells are numbered

from top to bottom first::

0 4 8 12 -> 0 4 8 12 -- 1st row doesn't change

1 5 9 13 -> 5 9 13 1 -- row shifted to left by 1 (wraps around)

2 6 10 14 -> 10 14 2 6 -- shifted by 2

3 7 11 15 -> 15 3 7 11 -- shifted by 3

"""

b[1], b[5], b[9], b[13] = b[5], b[9], b[13], b[1]

b[2], b[6], b[10], b[14] = b[10], b[14], b[2], b[6]

b[3], b[7], b[11], b[15] = b[15], b[3], b[7], b[11]

#print 'ShiftRows :', b

def shift\_rows\_inv(self, b):

"""

Similar to shift\_rows above, but performed in inverse for decryption.

"""

b[5], b[9], b[13], b[1] = b[1], b[5], b[9], b[13]

b[10], b[14], b[2], b[6] = b[2], b[6], b[10], b[14]

b[15], b[3], b[7], b[11] = b[3], b[7], b[11], b[15]

#print 'ShiftRows :', b

def mix\_columns(self, block):

"""MixColumns step. Mixes the values in each column"""

# Cache global multiplication tables (see below)

mul\_by\_2 = gf\_mul\_by\_2

mul\_by\_3 = gf\_mul\_by\_3

# Since we're dealing with a transposed matrix, columns are already

# sequential

for col in xrange(0, 16, 4):

v0, v1, v2, v3 = block[col:col + 4]

block[col] = mul\_by\_2[v0] ^ v3 ^ v2 ^ mul\_by\_3[v1]

block[col + 1] = mul\_by\_2[v1] ^ v0 ^ v3 ^ mul\_by\_3[v2]

block[col + 2] = mul\_by\_2[v2] ^ v1 ^ v0 ^ mul\_by\_3[v3]

block[col + 3] = mul\_by\_2[v3] ^ v2 ^ v1 ^ mul\_by\_3[v0]

#print 'MixColumns :', block

def mix\_columns\_inv(self, block):

"""

Similar to mix\_columns above, but performed in inverse for decryption.

"""

# Cache global multiplication tables (see below)

mul\_9 = gf\_mul\_by\_9

mul\_11 = gf\_mul\_by\_11

mul\_13 = gf\_mul\_by\_13

mul\_14 = gf\_mul\_by\_14

# Since we're dealing with a transposed matrix, columns are already

# sequential

for col in xrange(0, 16, 4):

v0, v1, v2, v3 = block[col:col + 4]

block[col] = mul\_14[v0] ^ mul\_9[v3] ^ mul\_13[v2] ^ mul\_11[v1]

block[col + 1] = mul\_14[v1] ^ mul\_9[v0] ^ mul\_13[v3] ^ mul\_11[v2]

block[col + 2] = mul\_14[v2] ^ mul\_9[v1] ^ mul\_13[v0] ^ mul\_11[v3]

block[col + 3] = mul\_14[v3] ^ mul\_9[v2] ^ mul\_13[v1] ^ mul\_11[v0]

#print 'MixColumns :', block

def encrypt\_block(self, block):

"""Encrypts a single block. This is the main AES function"""

# For efficiency reasons, the state between steps is transmitted via a

# mutable array, not returned

self.add\_round\_key(block, 0)

for round in xrange(1, self.rounds):

self.sub\_bytes(block, aes\_sbox)

self.shift\_rows(block)

self.mix\_columns(block)

self.add\_round\_key(block, round)

self.sub\_bytes(block, aes\_sbox)

self.shift\_rows(block)

# no mix\_columns step in the last round

self.add\_round\_key(block, self.rounds)

def decrypt\_block(self, block):

"""Decrypts a single block. This is the main AES decryption function"""

# For efficiency reasons, the state between steps is transmitted via a

# mutable array, not returned

self.add\_round\_key(block, self.rounds)

# count rounds down from (self.rounds) ... 1

for round in xrange(self.rounds - 1, 0, -1):

self.shift\_rows\_inv(block)

self.sub\_bytes(block, aes\_inv\_sbox)

self.add\_round\_key(block, round)

self.mix\_columns\_inv(block)

self.shift\_rows\_inv(block)

self.sub\_bytes(block, aes\_inv\_sbox)

self.add\_round\_key(block, 0)

# no mix\_columns step in the last round

#### ECB mode implementation

class ECBMode(object):

"""Electronic CodeBook (ECB) mode encryption.

Basically this mode applies the cipher function to each block individually;

no feedback is done. NB! This is insecure for almost all purposes

"""

def \_\_init\_\_(self, cipher):

self.cipher = cipher

self.block\_size = cipher.block\_size

def ecb(self, data, block\_func):

"""Perform ECB mode with the given function"""

if len(data) % self.block\_size != 0:

raise ValueError("Input length must be multiple of 16")

block\_size = self.block\_size

data = array('B', data)

for offset in xrange(0, len(data), block\_size):

block = data[offset:offset + block\_size]

block\_func(block)

data[offset:offset + block\_size] = block

return data.tostring()

def encrypt(self, data):

"""Encrypt data in ECB mode"""

return self.ecb(data, self.cipher.encrypt\_block)

def decrypt(self, data):

"""Decrypt data in ECB mode"""

return self.ecb(data, self.cipher.decrypt\_block)

#### CBC mode

class CBCMode(object):

"""

Cipher Block Chaining(CBC) mode encryption. This mode avoids content leaks.

In CBC encryption, each plaintext block is XORed with the ciphertext block

preceding it; decryption is simply the inverse.

"""

# A better explanation of CBC can be found here:

# http://en.wikipedia.org/wiki/Block\_cipher\_modes\_of\_operation#-

# Cipher-block\_chaining\_.28CBC.29

def \_\_init\_\_(self, cipher, IV):

self.cipher = cipher

self.block\_size = cipher.block\_size

self.IV = array('B', IV)

def encrypt(self, data):

"""Encrypt data in CBC mode"""

block\_size = self.block\_size

if len(data) % block\_size != 0:

raise ValueError("Plaintext length must be multiple of 16")

data = array('B', data)

IV = self.IV

for offset in xrange(0, len(data), block\_size):

block = data[offset:offset + block\_size]

# Perform CBC chaining

for i in xrange(block\_size):

block[i] ^= IV[i]

self.cipher.encrypt\_block(block)

data[offset:offset + block\_size] = block

IV = block

self.IV = IV

return data.tostring()

def decrypt(self, data):

"""Decrypt data in CBC mode"""

block\_size = self.block\_size

if len(data) % block\_size != 0:

raise ValueError("Ciphertext length must be multiple of 16")

data = array('B', data)

IV = self.IV

for offset in xrange(0, len(data), block\_size):

ctext = data[offset:offset + block\_size]

block = ctext[:]

self.cipher.decrypt\_block(block)

# Perform CBC chaining

#for i in xrange(block\_size):

# data[offset + i] ^= IV[i]

for i in xrange(block\_size):

block[i] ^= IV[i]

data[offset:offset + block\_size] = block

IV = ctext

#data[offset : offset+block\_size] = block

self.IV = IV

return data.tostring()

def galois\_multiply(a, b):

"""Galois Field multiplicaiton for AES"""

p = 0

while b:

if b & 1:

p ^= a

a <<= 1

if a & 0x100:

a ^= 0x1b

b >>= 1

return p & 0xff

# Precompute the multiplication tables for encryption

gf\_mul\_by\_2 = array('B', [galois\_multiply(x, 2) for x in range(256)])

gf\_mul\_by\_3 = array('B', [galois\_multiply(x, 3) for x in range(256)])

# ... for decryption

gf\_mul\_by\_9 = array('B', [galois\_multiply(x, 9) for x in range(256)])

gf\_mul\_by\_11 = array('B', [galois\_multiply(x, 11) for x in range(256)])

gf\_mul\_by\_13 = array('B', [galois\_multiply(x, 13) for x in range(256)])

gf\_mul\_by\_14 = array('B', [galois\_multiply(x, 14) for x in range(256)])

####

# The S-box is a 256-element array, that maps a single byte value to another

# byte value. Since it's designed to be reversible, each value occurs only once

# in the S-box

#

# More information: http://en.wikipedia.org/wiki/Rijndael\_S-box

aes\_sbox = array(

'B',

'637c777bf26b6fc53001672bfed7ab76'

'ca82c97dfa5947f0add4a2af9ca472c0'

'b7fd9326363ff7cc34a5e5f171d83115'

'04c723c31896059a071280e2eb27b275'

'09832c1a1b6e5aa0523bd6b329e32f84'

'53d100ed20fcb15b6acbbe394a4c58cf'

'd0efaafb434d338545f9027f503c9fa8'

'51a3408f929d38f5bcb6da2110fff3d2'

'cd0c13ec5f974417c4a77e3d645d1973'

'60814fdc222a908846eeb814de5e0bdb'

'e0323a0a4906245cc2d3ac629195e479'

'e7c8376d8dd54ea96c56f4ea657aae08'

'ba78252e1ca6b4c6e8dd741f4bbd8b8a'

'703eb5664803f60e613557b986c11d9e'

'e1f8981169d98e949b1e87e9ce5528df'

'8ca1890dbfe6426841992d0fb054bb16'.decode('hex')

)

# This is the inverse of the above. In other words:

# aes\_inv\_sbox[aes\_sbox[val]] == val

aes\_inv\_sbox = array(

'B',

'52096ad53036a538bf40a39e81f3d7fb'

'7ce339829b2fff87348e4344c4dee9cb'

'547b9432a6c2233dee4c950b42fac34e'

'082ea16628d924b2765ba2496d8bd125'

'72f8f66486689816d4a45ccc5d65b692'

'6c704850fdedb9da5e154657a78d9d84'

'90d8ab008cbcd30af7e45805b8b34506'

'd02c1e8fca3f0f02c1afbd0301138a6b'

'3a9111414f67dcea97f2cfcef0b4e673'

'96ac7422e7ad3585e2f937e81c75df6e'

'47f11a711d29c5896fb7620eaa18be1b'

'fc563e4bc6d279209adbc0fe78cd5af4'

'1fdda8338807c731b11210592780ec5f'

'60517fa919b54a0d2de57a9f93c99cef'

'a0e03b4dae2af5b0c8ebbb3c83539961'

'172b047eba77d626e169146355210c7d'.decode('hex')

)

# The Rcon table is used in AES's key schedule (key expansion)

# It's a pre-computed table of exponentation of 2 in AES's finite field

#

# More information: http://en.wikipedia.org/wiki/Rijndael\_key\_schedule

aes\_Rcon = array(

'B',

'8d01020408102040801b366cd8ab4d9a'

'2f5ebc63c697356ad4b37dfaefc59139'

'72e4d3bd61c29f254a943366cc831d3a'

'74e8cb8d01020408102040801b366cd8'

'ab4d9a2f5ebc63c697356ad4b37dfaef'

'c5913972e4d3bd61c29f254a943366cc'

'831d3a74e8cb8d01020408102040801b'

'366cd8ab4d9a2f5ebc63c697356ad4b3'

'7dfaefc5913972e4d3bd61c29f254a94'

'3366cc831d3a74e8cb8d010204081020'

'40801b366cd8ab4d9a2f5ebc63c69735'

'6ad4b37dfaefc5913972e4d3bd61c29f'

'254a943366cc831d3a74e8cb8d010204'

'08102040801b366cd8ab4d9a2f5ebc63'

'c697356ad4b37dfaefc5913972e4d3bd'

'61c29f254a943366cc831d3a74e8cb'.decode('hex')

)