SIMON Encryption and Decryption

CODE:

```
from __future__ import print_function
from collections import deque


class SimonCipher(object):
    """Simon Block Cipher Object"""

    #The constant sequence, z_x is created by a Linear Feedback Shift Register (LFSR). The logical
sequence of bit constants is set by the value of the key and block sizes. The LFSR is created by a 5-bit
field. The constant bit operates on a key block once per round on the lowest bit in order to add non-
key-dependent entropy to the key schedule. The LFSR has different logic for each z_x sequence;
however, the initial condition is the same for encryption. The initial condition of the LFSR for
decryption varies on the round. z_0, z_1, z_2, z_3 and z_4 are the constant sequence. (Underscore is
used to represent the subscript)

    # Z Arrays (stored bit reversed for easier usage)
    z0 = 0b01100111000011010100100010111110110011100001101010010001011111
    z1 = 0b01011010000110010011111011100010101101000011001001111101110001
    z2 = 0b11001101101001111110001000010100011001001011000000111011110101
    z3 = 0b11110000101100111001010001001000000111101001100011010111011011
    z4 = 0b11110111001001010011000011101000000100011011010110011110001011

    # valid cipher configurations stored:
    # block_size:{key_size:(number_rounds,z sequence)}
    __valid_setups = {32: {64: (32, z0)},
                      48: {72: (36, z0), 96: (36, z1)},
                      64: {96: (42, z2), 128: (44, z3)},
                      96: {96: (52, z2), 144: (54, z3)},
                      128: {128: (68, z2), 192: (69, z3), 256: (72, z4)}}

    # the valid setup is done as per the protocols of the block size, key size and the number of rounds
    # there are different modes that can be used for the implementation of the Simon Cipher like
Electronic Code Book ECB, which is also the Default mode, Counter CTR, Cipher Block Chaining
CBC, Propagating Cipher Block Chaining PCBC, Cipher Feedback CFB, Output Feedback OFB.
Here we implement only using the ECB which is also it default mode of operation.
    __valid_modes = ['ECB']

    def __init__(self, key, key_size=128, block_size=128, mode='ECB', init=0, counter=0):
        """
        Initialize an instance of the Simon block cipher.
        :param key: Int representation of the encryption key
        :param key_size: Int representing the encryption key in bits
        :param block_size: Int representing the block size in bits
        :param mode: String representing which cipher block mode the object should initialize with
        :param init: IV for CTR, CBC, PCBC, CFB, and OFB modes. Here we use only Electronic Code
Book
        :param counter: Initial Counter value
        :return: None
        """

        # Setup block/word size
```

```
try:
    self.possible_setups = self.__valid_setups[block_size]
    self.block_size = block_size
    self.word_size = self.block_size >> 1
except KeyError:
    print('Invalid block size!')
    print('Please use one of the following block sizes:', [x for x in self.__valid_setups.keys()])
    raise

# Setup Number of Rounds, Z Sequence, and Key Size
try:
    self.rounds, self.zseq = self.possible_setups[key_size]
    self.key_size = key_size
except KeyError:
    print('Invalid key size for selected block size!!')
    print('Please use one of the following key sizes:', [x for x in self.possible_setups.keys()])
    raise

# Create Properly Sized bit mask for truncating addition and left shift outputs
self.mod_mask = (2 ** self.word_size) - 1

# Parse the given iv and truncate it to the block length
try:
    self.iv = init & ((2 ** self.block_size) - 1)
    self.iv_upper = self.iv >> self.word_size
    self.iv_lower = self.iv & self.mod_mask
except (ValueError, TypeError):
    print('Invalid IV Value!')
    print('Please Provide IV as int')
    raise

# Parse the given Counter and truncate it to the block length
try:
    self.counter = counter & ((2 ** self.block_size) - 1)
except (ValueError, TypeError):
    print('Invalid Counter Value!')
    print('Please Provide Counter as int')
    raise

# Check Cipher Mode
try:
    position = self.__valid_modes.index(mode)
    self.mode = self.__valid_modes[position]
except ValueError:
    print('Invalid cipher mode!')
    print('Please use one of the following block cipher modes:', self.__valid_modes)
    raise

# Parse the given key and truncate it to the key length
try:
    self.key = key & ((2 ** self.key_size) - 1)
except (ValueError, TypeError):
    print('Invalid Key Value!')
    print('Please Provide Key as int')
    raise
```

```
    # Pre-compile key schedule
    m = self.key_size // self.word_size
    self.key_schedule = []

    # Create list of subwords from encryption key
    k_init = [((self.key >> (self.word_size * ((m-1) - x))) & self.mod_mask) for x in range(m)]

    k_reg = deque(k_init)  # Use queue to manage key subwords

    round_constant = self.mod_mask ^ 3  # Round Constant is 0xFFFF..FC

    # Generate all round keys
    for x in range(self.rounds):

        rs_3 = ((k_reg[0] << (self.word_size - 3)) + (k_reg[0] >> 3)) & self.mod_mask

        if m == 4:
            rs_3 = rs_3 ^ k_reg[2]

        rs_1 = ((rs_3 << (self.word_size - 1)) + (rs_3 >> 1)) & self.mod_mask

        c_z = ((self.zseq >> (x % 62)) & 1) ^ round_constant

        new_k = c_z ^ rs_1 ^ rs_3 ^ k_reg[m - 1]

        self.key_schedule.append(k_reg.pop())
        k_reg.appendleft(new_k)

def encrypt_round(self, x, y, k):
    """
    Complete One Feistel Round
    :param x: Upper bits of current plaintext
    :param y: Lower bits of current plaintext
    :param k: Round Key
    :return: Upper and Lower ciphertext segments
    """

    # Generate all circular shifts
    ls_1_x = ((x >> (self.word_size - 1)) + (x << 1)) & self.mod_mask
    ls_8_x = ((x >> (self.word_size - 8)) + (x << 8)) & self.mod_mask
    ls_2_x = ((x >> (self.word_size - 2)) + (x << 2)) & self.mod_mask

    # XOR Chain
    xor_1 = (ls_1_x & ls_8_x) ^ y
    xor_2 = xor_1 ^ ls_2_x
    new_x = k ^ xor_2

    return new_x, x

def decrypt_round(self, x, y, k):
    """Complete One Inverse Feistel Round
    :param x: Upper bits of current ciphertext
    :param y: Lower bits of current ciphertext
    :param k: Round Key
```

```
        :return: Upper and Lower plaintext segments
        """

        # Generate all circular shifts
        ls_1_y = ((y >> (self.word_size - 1)) + (y << 1)) & self.mod_mask
        ls_8_y = ((y >> (self.word_size - 8)) + (y << 8)) & self.mod_mask
        ls_2_y = ((y >> (self.word_size - 2)) + (y << 2)) & self.mod_mask

        # Inverse XOR Chain
        xor_1 = k ^ x
        xor_2 = xor_1 ^ ls_2_y
        new_x = (ls_1_y & ls_8_y) ^ xor_2

        return y, new_x

    def encrypt(self, plaintext):
        """
        Process new plaintext into ciphertext based on current cipher object setup
        :param plaintext: Int representing value to encrypt
        :return: Int representing encrypted value
        """
        try:
            b = (plaintext >> self.word_size) & self.mod_mask
            a = plaintext & self.mod_mask
        except TypeError:
            print('Invalid plaintext!')
            print('Please provide plaintext as int')
            raise

        if self.mode == 'ECB':
            b, a = self.encrypt_function(b, a)

        ciphertext = (b << self.word_size) + a

        return ciphertext

    def decrypt(self, ciphertext):
        """
        Process new ciphertest into plaintext based on current cipher object setup
        :param ciphertext: Int representing value to encrypt
        :return: Int representing decrypted value
        """
        try:
            b = (ciphertext >> self.word_size) & self.mod_mask
            a = ciphertext & self.mod_mask
        except TypeError:
            print('Invalid ciphertext!')
            print('Please provide ciphertext as int')
            raise

        if self.mode == 'ECB':
            a, b = self.decrypt_function(a, b)

        plaintext = (b << self.word_size) + a
```

```
    return plaintext


def encrypt_function(self, upper_word, lower_word):
    """
    Completes appropriate number of Simon Fiestel function to encrypt provided words
    Round number is based off of number of elements in key schedule
    upper_word: int of upper bytes of plaintext input
                limited by word size of currently configured cipher
    lower_word: int of lower bytes of plaintext input
                limited by word size of currently configured cipher
    x,y:        int of Upper and Lower ciphertext words
    """
    x = upper_word
    y = lower_word

    # Run Encryption Steps For Appropriate Number of Rounds
    for k in self.key_schedule:
         # Generate all circular shifts
        ls_1_x = ((x >> (self.word_size - 1)) + (x << 1)) & self.mod_mask
        ls_8_x = ((x >> (self.word_size - 8)) + (x << 8)) & self.mod_mask
        ls_2_x = ((x >> (self.word_size - 2)) + (x << 2)) & self.mod_mask

        # XOR Chain
        xor_1 = (ls_1_x & ls_8_x) ^ y
        xor_2 = xor_1 ^ ls_2_x
        y = x
        x = k ^ xor_2

    return x,y

def decrypt_function(self, upper_word, lower_word):
    """
    Completes appropriate number of Simon Fiestel function to decrypt provided words
    Round number is based off of number of elements in key schedule
    upper_word: int of upper bytes of ciphertext input
                limited by word size of currently configured cipher
    lower_word: int of lower bytes of ciphertext input
                limited by word size of currently configured cipher
    x,y:        int of Upper and Lower plaintext words
    """
    x = upper_word
    y = lower_word

    # Run Encryption Steps For Appropriate Number of Rounds
    for k in reversed(self.key_schedule):
         # Generate all circular shifts
        ls_1_x = ((x >> (self.word_size - 1)) + (x << 1)) & self.mod_mask
        ls_8_x = ((x >> (self.word_size - 8)) + (x << 8)) & self.mod_mask
        ls_2_x = ((x >> (self.word_size - 2)) + (x << 2)) & self.mod_mask

        # XOR Chain
        xor_1 = (ls_1_x & ls_8_x) ^ y
        xor_2 = xor_1 ^ ls_2_x
        y = x
```

```
        x = k ^ xor_2

    return x,y




if __name__ == "__main__":
    #initialising the cipher object with encryption key
    cipher = SimonCipher(0x1918111009080100, key_size=64, block_size=32)
    #Say, we want to encrypt 273.
    #We have thus entered 0x111, which is the hexadecimal value for 273
    #encrypt() is the function by which the plaintext gets converted to ciphertext
    t = cipher.encrypt(0x111)
    print("Encrypted Message in Hex Form")
    print(hex(t))
    #the encrypted message was displayed in hex form
    #decrypt() is the function by which the ciphertext gets converted to plaintext
    z = cipher.decrypt(t)
    print("Decrypted Message in Hex form")
    print(hex(z))
    #the decrypted message was displayed in hex form
```

OUTPUT

```
aritra@aritra:~/Aritra_SIM_SPK$ python3 simon.py
Encrypted Message in Hex Form
0x55ea5ec1
Decrypted Message in Hex form
0x111
aritra@aritra:~/Aritra_SIM_SPK$
```

SPECK Encryption and Decryption

CODE:

```
from __future__ import print_function


class SpeckCipher(object):
    """Speck Block Cipher Object"""
    # valid cipher configurations stored:
    # block_size:{key_size:number_rounds}
    __valid_setups = {32: {64: 22},
                      48: {72: 22, 96: 23},
                      64: {96: 26, 128: 27},
                      96: {96: 28, 144: 29},
                      128: {128: 32, 192: 33, 256: 34}}
    # the valid setup is done as per the protocols of the block size, key size and the number of rounds
    # there are different modes that can be used for the implementation of the Speck Cipher like
Electronic Code Book ECB, which is also the Default mode, Counter CTR, Cipher Block Chaining
CBC, Propagating Cipher Block Chaining PCBC, Cipher Feedback CFB, Output Feedback OFB.
Here we implement only using the ECB which is also it default mode of operation.

    __valid_modes = ['ECB']

    def encrypt_round(self, x, y, k):
        """Complete One Round of Feistel Operation"""
        rs_x = ((x << (self.word_size - self.alpha_shift)) + (x >> self.alpha_shift)) & self.mod_mask

        add_sxy = (rs_x + y) & self.mod_mask

        new_x = k ^ add_sxy

        ls_y = ((y >> (self.word_size - self.beta_shift)) + (y << self.beta_shift)) & self.mod_mask

        new_y = new_x ^ ls_y

        return new_x, new_y

    def decrypt_round(self, x, y, k):
        """Complete One Round of Inverse Feistel Operation"""

        xor_xy = x ^ y

        new_y = ((xor_xy << (self.word_size - self.beta_shift)) + (xor_xy >> self.beta_shift)) &
self.mod_mask

        xor_xk = x ^ k

        msub = ((xor_xk - new_y) + self.mod_mask_sub) % self.mod_mask_sub

        new_x = ((msub >> (self.word_size - self.alpha_shift)) + (msub << self.alpha_shift)) &
self.mod_mask

        return new_x, new_y
```

```
def __init__(self, key, key_size=128, block_size=128, mode='ECB', init=0, counter=0):

    # Setup block/word size
    try:
        self.possible_setups = self.__valid_setups[block_size]
        self.block_size = block_size
        self.word_size = self.block_size >> 1
    except KeyError:
        print('Invalid block size!')
        print('Please use one of the following block sizes:', [x for x in self.__valid_setups.keys()])
        raise

    # Setup Number of Rounds and Key Size
    try:
        self.rounds = self.possible_setups[key_size]
        self.key_size = key_size
    except KeyError:
        print('Invalid key size for selected block size!!')
        print('Please use one of the following key sizes:', [x for x in self.possible_setups.keys()])
        raise

    # Create Properly Sized bit mask for truncating addition and left shift outputs
    self.mod_mask = (2 ** self.word_size) - 1

    # Mod mask for modular subtraction
    self.mod_mask_sub = (2 ** self.word_size)

    # Setup Circular Shift Parameters
    if self.block_size == 32:
        self.beta_shift = 2
        self.alpha_shift = 7
    else:
        self.beta_shift = 3
        self.alpha_shift = 8

    # Parse the given iv and truncate it to the block length
    try:
        self.iv = init & ((2 ** self.block_size) - 1)
        self.iv_upper = self.iv >> self.word_size
        self.iv_lower = self.iv & self.mod_mask
    except (ValueError, TypeError):
        print('Invalid IV Value!')
        print('Please Provide IV as int')
        raise

    # Parse the given Counter and truncate it to the block length
    try:
        self.counter = counter & ((2 ** self.block_size) - 1)
    except (ValueError, TypeError):
        print('Invalid Counter Value!')
        print('Please Provide Counter as int')
        raise

    # Check Cipher Mode
    try:
```

```
            position = self.__valid_modes.index(mode)
            self.mode = self.__valid_modes[position]
        except ValueError:
            print('Invalid cipher mode!')
            print('Please use one of the following block cipher modes:', self.__valid_modes)
            raise

        # Parse the given key and truncate it to the key length
        try:
            self.key = key & ((2 ** self.key_size) - 1)
        except (ValueError, TypeError):
            print('Invalid Key Value!')
            print('Please Provide Key as int')
            raise

        # Pre-compile key schedule
        self.key_schedule = [self.key & self.mod_mask]
        l_schedule = [(self.key >> (x * self.word_size)) & self.mod_mask for x in
                      range(1, self.key_size // self.word_size)]

        for x in range(self.rounds - 1):
            new_l_k = self.encrypt_round(l_schedule[x], self.key_schedule[x], x)
            l_schedule.append(new_l_k[0])
            self.key_schedule.append(new_l_k[1])

    def encrypt(self, plaintext):
        try:
            b = (plaintext >> self.word_size) & self.mod_mask
            a = plaintext & self.mod_mask
        except TypeError:
            print('Invalid plaintext!')
            print('Please provide plaintext as int')
            raise

        if self.mode == 'ECB':
            b, a = self.encrypt_function(b, a)

        ciphertext = (b << self.word_size) + a

        return ciphertext

    def decrypt(self, ciphertext):
        try:
            b = (ciphertext >> self.word_size) & self.mod_mask
            a = ciphertext & self.mod_mask
        except TypeError:
            print('Invalid ciphertext!')
            print('Please provide plaintext as int')
            raise

        if self.mode == 'ECB':
            b, a = self.decrypt_function(b, a)

        plaintext = (b << self.word_size) + a
```

```
        return plaintext

    def encrypt_function(self, upper_word, lower_word):

        x = upper_word
        y = lower_word

        # Run Encryption Steps For Appropriate Number of Rounds
        for k in self.key_schedule:
            rs_x = ((x << (self.word_size - self.alpha_shift)) + (x >> self.alpha_shift)) & self.mod_mask

            add_sxy = (rs_x + y) & self.mod_mask

            x = k ^ add_sxy

            ls_y = ((y >> (self.word_size - self.beta_shift)) + (y << self.beta_shift)) & self.mod_mask

            y = x ^ ls_y

        return x,y


    def decrypt_function(self, upper_word, lower_word):

        x = upper_word
        y = lower_word

        # Run Encryption Steps For Appropriate Number of Rounds
        for k in reversed(self.key_schedule):
            xor_xy = x ^ y

            y = ((xor_xy << (self.word_size - self.beta_shift)) + (xor_xy >> self.beta_shift)) &
self.mod_mask

            xor_xk = x ^ k

            msub = ((xor_xk - y) + self.mod_mask_sub) % self.mod_mask_sub

            x = ((msub >> (self.word_size - self.alpha_shift)) + (msub << self.alpha_shift)) &
self.mod_mask


        return x,y



if __name__ == "__main__":
    #initialising the cipher object with encryption key
    cipher =
SpeckCipher(0x1f1e1d1c1b1a191817161514131211100f0e0d0c0b0a09080706050403020100, 256,
128, 'ECB')
    #Say, we want to encrypt 273.
    #We have thus entered 0x111, which is the hexadecimal value for 273
    #encrypt() is the function by which the plaintext gets converted to ciphertext
    t = cipher.encrypt(0x111)
```

```
print("Encrypted Message in Hex Form")
print(hex(t))
#the encrypted message was displayed in hex form
#decrypt() is the function by which the ciphertext gets converted to plaintext
z = cipher.decrypt(t)
print("Decrypted Message in Hex form")
print(hex(z))
#the decrypted message was displayed in hex form
```

OUTPUT

```
aritra@aritra:~/Aritra_SIM_SPK$ python3 speck.py
Encrypted Message in Hex Form
0xf8d336f8e89c517e8543988a7d9da0ae
Decrypted Message in Hex form
0x111
aritra@aritra:~/Aritra_SIM_SPK$
```