AES

Step 1. Define Lists, that Will Be Used in Calculations

In order to perform AES encryption some static lists are needed. Note that numbers in these lists are in written in hex, which in Python is marked as integers starting with *0x*, for instance, 0x10 is equal to 16 in decimal.

These are the lists used in AES:

- RCON array (variable *RCON*) is used to get values for the T function. Since in this example AES will have 10 rounds, the RCON array will have 10 elements.

  Python implementation:

  ```
  RCON = [0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36]
  ```

- S-box (variable *SBOX*), which is 16 x 16 matrix of values which will be used in substitution process, which is similar to the one in DES.

  Python implementation:

```
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]]
```

- Mix matrix (variable *MIX_MATRIX*) which is 4 x 4 matrix with values from 1 to 3 (value 1 is repeated twice) and is used for mixing columns.

  Python implementation:

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

Step 2. Define T function

T function is used for every subkey with index that is a multiple of 4 and consists of performing S-box substitution for sub key after single left shift and then performing XOR operation between first value in the new array with corresponding value in the RCON table (since key generation counts sub keys, the correct value of RCON array elopement is calculated by dividing the subkey index and subtracting 1 as the arrays are counted from 0 in Python)

Python implementation of T function:

```
def t_function(self, w, i):
    # Substitute bytes after performing single left shift:
    w = [self.get_sbox_value(foo) for foo in numpy.roll(w, -1)]
    # Perform XOR operation between 1st value and round constant:
    # Round constant value is taken from RCON list
    w[0] = int(w[0]) ^ RCON[int(i / 4) - 1]
    return w
```

Substitution is achieved by selecting values by dividing passed value by 0x10 (or 16 in decimal) and replacing it with the one from the S-box where x axis is defined by the quotient and y axis by remainder.

Python Implementation of substitution:
```
def get_sbox_value(number):
        return SBOX[int(number / 0x10)][int(number % 0x10)]
```

Step 3. Generate Round Keys

As mentioned before, our AES implementation will have 10 rounds so 10 round keys will have to be generated. Each round key is put together from 4 parts that are processed by key function. Initial values for round key generation are extracted by splitting key into 4 parts. Then each other subkey (there are going to be 40 of them) are generated by the following function where T is the T function:

If i is a multiple of 4:

$W(i) = W(i\text{-}4) \oplus T(W(i\text{-}1))$

Else:

$W(i) = W(i\text{-}4) \oplus W(i\text{-}1)$

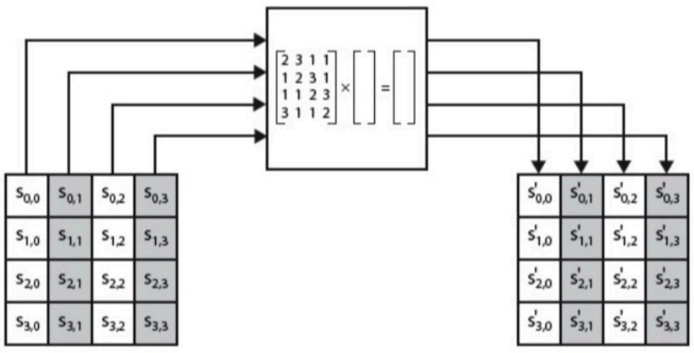
After calculating all 40 subkeys, the keys are returned as nested array with 11 members (initial key plus 10 round keys) with 4 elements each (the 4 subkeys)

Python implementation for key generation:
```
def generate_round_keys(self, key):
    # Get initial round 4 keys by splitting main key:
    all_keys = self.split_array(key, 4)
    # Perform operations for the next 40 keys:
    for i in range(4, 44):
        # Get 1st key for new key generation:
        w1 = all_keys[i - 4]
        # Get 2nd key for key generation:
        # If it's sequence number is 4 multiple, perform T function on it,
        # else select key according to the rules
        w2 = self.t_function(all_keys[i - 1], i) if i % 4 == 0
             else all_keys[i - 1]

        all_keys.append([w1[foo] ^ w2[foo] for foo in range(4)])
    return self.split_array(all_keys, 4)
```

Step 4. Define Column Mixing Function

Column mixing is achieved by performing special bit multiplication between corresponding values of passed message block matrix and mixing matrix (variable *MIX_COLUMNS*). Image below illustrates how mixing is done.



2 3 1 1
1 2 3 1
1 1 2 3
3 1 1 2
×
=
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,1}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$

Python implementation for mixing columns:

```python
def mix_columns(self, text):
    # Define new 4x4 matrix to store values
    mixed_columns = [[0 for a in range(4)] for b in range(4)]
    # Fill matrix by going through text and mix arrays and performing AES
    # multiplication operation
    for col in range(4):
        for row in range(4):
            for i in [self.bit_multiplication(text[i][col],
                      MIX_MATRIX[row][i]) for i in range(4)]:
                mixed_columns[row][col] ^= i
```

AES bit multiplication can be achieved by using complicated polynomial arithmetic, but since there are only 3 different of multipliers, it can be put into if else statement following these steps:

- Number from message block should be converted to 8-bit binary number, then the resulting binary should be shifted one bit to the left and have 0 added to the end
- Then depending of the multiplier, result can get the following values:
    - **If multiplier is equal to 2**, result can be as a 1-bit left shift, if the leftmost bit of the original value (before the shift) is 1, it should be followed by a conditional bitwise XOR with (0001 1011).
    - **If multiplier is equal to 3**, result will be equal to result of conditional bitwise XOR result of value bit multiplication by 2 and the number.
    - **If multiplier is equal to 1**, the value remains the same.

Python implementation of bit multiplication:

```python
def bit_multiplication(number, multiplier):

    # Convert number to 8bit string
    num_string = '{0:08b}'.format(number)
    # Perform left shift and add 0 to the end
    new_num = int(num_string[1:] + '0', 2)

    if multiplier == 0x02:
# Multiplication of a value by 2 can be implemented as a 1-bit left shift,
# if the leftmost bit of the original value (before the shift) is 1,
# it should be followed by a conditional bitwise XOR with (0001 1011)
        return new_num ^ 0b11011 if num_string[0] == '1' else new_num

    elif multiplier == 0x03:
        # Multiplication by 3 can be achieved by performing XOR operation
        # between value of number multiplied by 2 and original value
        return (new_num ^ 0b11011 if num_string[0] == '1'
               else new_num) ^ number
    else:
        # Multiplication by 1 leaves number unchanged
        return number
```

Step 5. Perform AES Rounds

Before applying 10 AES rounds for a message to create ciphertext, all round keys are generated, then message is split to 4 blocks with 4 hex numbers each (both key and message are fixed length of 16 hex numbers in this example) and then message blocks are flipped (rows are changed with columns), then round key is applied (round key application consists of XOR operation between corresponding members of passed block and flipped round key matrix).

These actions give initial value for *block* variable, that will be used during rounds and will be eventually converted to ciphertext.

Python implementation:
```
keys = self.generate_round_keys(key)
message_blocks = self.flip_matrix(self.split_array(message, 4))
block = self.apply_round_key(message_blocks, keys[0])
```

Python implementation of matrix flipping:
```
def flip_matrix(key):
    # Switches rows with columns in 4x4 matrix
    new_key = [[0 for foo in range(4)] for bar in range(4)]
    for foo in range(len(key)):
        for bar in range(len(key[foo % 4])):
            new_key[bar][foo % 4] = key[foo % 4][bar]
    return new_key
```

Python implementation of round key application:
```
def apply_round_key(self, cipher, round_key):
    # Perform XOR operations between corresponding
    # members in cipher and round key:
    return [[cipher[foo][bar] ^
             self.flip_matrix(round_key)[foo][bar]
            for bar in range(4)] for foo in range(4)]
```

- AES round consists of the following procedures:
    - Value substitution with corresponding ones from the S-box

      Python implementation:
      ```
      block = [[self.get_sbox_value(foo) for foo in bar]
               for bar in block]
      ```
    - Shift rows cyclically to the left by offsets of 0, 1, 2 and 3 for lines 1 to 4. This was achieved by reiterating through the matrix and multiplying -1 (which indicates single left shift in *numpy.roll* method) and multiplying it by i which is line or sub array index and since the enumeration of arrays starts from 0, this allows efficiently cycle through array and achieve desired left shifts for all the lines.

      Python implementation:
      ```
      block = [[int(foo) for foo in numpy.roll(block[i], -1 * i)]
               for i in range(len(block))]
      ```
    - Columns for rounds 1 to 9 are also mixed according to the method described in Step 4.

      Python implementation:
      ```
      if round_no < 10:
          block = self.mix_columns(block)
      ```
- After all rounds are completed, the result array must be flipped in and arranged in a line which will create a ciphertext.

  Python implementation:
  ```
  output['cipher'] = self.format_output(self.flip_matrix(block),
                                        output_type='line')
  ```