Assignment 5

General Instructions:

1. Deadline: Monday, December 2nd at 11:59 pm.
2. This assignment is worth 10 pts of your course grade.
3. Download the assignment files, which consist of the following:
    - SDES_template.py (Rename to SDES.py)
    - test_A5.py (do not change)
    - utilities.py (do not change)
    - primes.txt, sbox1.txt and sbox2.txt (do not change)
4. Rename SDES_template.py to SDES.py. This is the only file that you will be editing for your solution.
5. Any use of external libraries will cause rejection of your assignment.
6. Submit only your SDES.py file.
7. Remember to put your name and ID on the top comment of the file.

Assignment Overview:

This entire assignment can be viewed as one single task which is to implement the Simple DES through specific configuration. This major task has been broken into five smaller tasks:

1- Encoding Scheme
2- Configuration
3- Key Generation
4- Feistel Cipher
5- Encryption/Decryption

The steps are incremental, i.e. you would need functions provided in previous steps to successfully complete the next step. Unit testing for each step, assumes that functions from previous steps are correct implementation.

The overall configuration of the SDES will be the following:

1- The B6 Encoding scheme will be used to represent plaintext and ciphertext
2- The block size is 12-bits, which is two B6 characters
3- The key size is 9-bits.
4- The number of rounds is 4
5- The key is generated through the Blum Blum Shub Algorithm
6- The Subkeys are generated through the circular left shift algorithm.
7- The Sbox and expand components of the F function are configured based on the block size.
8- The algorithm maintains a configuration file that defines the parameter values.

Q1: Encoding Scheme (2 pts)

The ASCII code uses one byte representation (0+ 7 bits) for each character. However, since the proposed SDES scheme uses 6 bits, we need another encoding scheme that works on 6 bits.

We will call the new scheme: "B6 Code". The scheme defines 64 symbols, which are the lower and upper case characters in addition to the ten numerical symbols. The full code can be obtained through `utilities.get_B6Code()`.

You need to create four functions:

```
1-encode(c, codeType)
2-decode(b, codeType)
3-encode_B6(c)
4-decode_B6(b)
```

The first two functions are two generic functions that control how encoding and decoding is performed. The encoding function receives a character and coding type. The two coding types we are interested in are the ASCII and B6. If an ASCII code is given, the function should return the corresponding 8-bit binary value, while if B6 is specified as the coding type, then 6-bit binary value should be returned. The decoding functions reverse the process, where binary value is

passed, along with the coding type, and the corresponding character is returned.

The implementation of the above two functions is generic and could be extended in the future. That is why, unless it is given an invalid value, it will consider other coding schemes, like Unicode, as unsupported.

Below are the function definitions:

```
#-------------------------------------------------------------
# Parameters:   c (str): a character
#               codeType (str)
# Return:       b (str): corresponding binary number
# Description:  Generic function for encoding
#               Current implementation supports only ASCII and B6 encoding
# Error:        If c is not a single character:
#                   print('Error(encode): invalid input'), return ''
#               If unsupported encoding type:
#                  print('Error(encode): Unsupported Coding Type'), return ''
#-------------------------------------------------------------
def encode(c,codeType):
    #your code here
    return b

#-------------------------------------------------------------
# Parameters:   b (str): a binary number
#               codeType (str)
# Return:       b (str): corresponding binary number
# Description:  Generic function for encoding
#               Current implementation supports only ASCII and B6 encoding
# Error:        If c is not a binary number:
#                   print('Error(decode): invalid input'), return ''
#               If unsupported encoding type:
#                  print('Error(decode): Unsupported Coding Type'), return ''
#-------------------------------------------------------------
def decode(b,codeType):
    #your code here
    return c
```

The third and fourth functions are the specific implementations of the B6 Coding scheme. Note that the ASCII encoding and decoding could be done in

separate functions that you add or inside the generic functions of encode/decode.

The two functions validate the input by ensuring that only a single character is passed to the encryption function and a valid 6-bit binary number is passed to the decryption function.

Below are the function definitions for the two functions:

```
#-------------------------------------------------------------
# Parameters:   c (str): a character
# Return:       B6Code (str): 6-digit binary code
# Description:  Encodes any given symbol in the B6 Encoding scheme
#               If given symbol is one of the 64 symbols, the function returns
#   the binary representation, which is the equivalent binary number of the
#   decimal value representing the position of the symbol in the B6Code
#               If the given symbol is not part of the B6Code -->
#                       return empty string (no error msg)
# Error:        If given input is not a single character -->
#                   print('Error(encode_B6): invalid input'), return ''
#-------------------------------------------------------------
def encode_B6(c):
    #your code here
    return b

#-------------------------------------------------------------
# Parameters:   b (str): binary number
# Return:       c (str): a character
# Description:  Decodes any given binary code in the B6 Coding scheme
#               Converts the binary number into integer, then get the
#               B6 code at that position
# Error:        If given input is not a valid 6-bit binary number -->
#                   print('Error(decode_B6): invalid input'), return ''
#-------------------------------------------------------------
def decode_B6(b):
    #your code here
    return c
```

Running the testing function will produce:

```
>>> test_q1()
-----------------------------------------
Testing Encoding:
encode('A','ASCII') =  01000001
encode('\n','ASCII') =  00001010
encode('A','B6') =  100100
encode('\n','B6') =  111111
encode('','B6') = Error(encode): invalid input
encode(1,'B6') = Error(encode): invalid input
encode('A','Unicode') = Error(encode): Unsupported Coding Type
encode_B6('a') =  001010
encode_B6('AB') = Error(encode_B6): invalid input

Testing Decoding:
decode('01000001','ASCII') =  A
decode('100100','B6') =  A
decode(1000001,'ASCII') = Error(decode): Invalid input
decode('01000001','Unicode') = Error(decode): Unsupported Coding Type
decode('','ASCII') = Error(decode): Invalid input
decode_B6('000001') =  1
decode_B6(100000) = Error(decode_B6): invalid input
decode_B6('0100000') = Error(decode_B6): invalid input

-----------------------------------------
```

Q2: SDES Configuration
(2 pts)

There are six configuration parameters for the SDES. These parameters are defined in get_SDES_parameters(), which are:

1-'encoding_type'
2-'block_size'
3-'key_size'
4-'rounds'
5-'p'
6-'q'

The last two parameters are used in the generation of the key by the Blum Blum Shub algorithm as outlined in Q3.

The configuration, i.e. pairs of parameter,value, are stored in the *configFile*. Other functions, developed in Q3-Q5, will access these parameters/values through the functions developed in this part.

You need to write three functions:

1- Function to retrieve current configuration (current parameter values): *get_SDES_config()*
2- Function to retrieve current value of a given configuration parameter *get_SDES_value(parameter)*
3- Function to set parameters: *config_SDES(parameter,value)*

The first function reads the current configuration from the *configFile*. The output is returned in a 2D list formatted as the following: [[parameter1,value1],[parameter2,value2],…]. If the configuration file is empty, then an empty list should be returned. The following is the function definition:

```
#-------------------------------------------------------------
# Parameters:   None
# Return:       configList (2D List)
# Description:  Returns the current configuraiton of SDES
#               configuration list is formatted as the following:
#               [[parameter1,value],[parameter2,value2],...]
#               The configurations are read from the configuration file
#               If configuration file is empty --> return []
# Error:        None
#-------------------------------------------------------------
def get_SDES_config():
    # your code here
    return configList
```

The second function receives a parameter from the user and returns the corresponding value as defined in the configuration file. If the given parameter is not defined in the file an error message is returned. However, if the given parameter is a valid parameter but has no corresponding value in the file, the function returns an empty string. Below is the function definition:

```
#-------------------------------------------------------------
# Parameters:   parameter (str)
# Return:       value (str)
# Description:  Returns the value of the parameter based on the current
# Error:        If the parameter is undefined in get_SDES_parameters() -->
```

```
#                   print('Error(get_SDES_value): invalid parameter'), return
''
#-------------------------------------------------------------
def get_SDES_value(parameter):
    # your code here
    return value
```

The last function attempts to set a parameter to a specific value. The function only works with the parameter list defined in *get_SDES_parameters()*. If another parameter is passed, an error message is printed. The only validation the function performs on the passed value is to check if it is a non-empty string. The function reads the contents of the configuration file, if the parameter already exist, then its value is updated to the new value. Otherwise, a new entry is added to the file with the following format: *parameterName:value*. The function returns True if the set operation was successful and False otherwise.

```
#-------------------------------------------------------------
# Parameters:   parameter (str)
#               value (str)
# Return:       None
# Description:  Sets an SDES parameter to the given value and stores
#               the output in the configuration file
#                  if the configuration file contains previous value for the
parameter
#               the function overrides it with the new value
#               otherwise, the new value is appended to the configuration file
# Error:        If the parameter is undefined in get_SDES_parameters() -->
#                   print('Error(cofig_SDES): invalid parameter')
#               If given value is not a string or is an empty string:
#                   print('Error(config_SDES): invalid value
#-------------------------------------------------------------
def config_SDES(parameter,value):
    # your code here
    return True
```

Running the testing module will give the following:

```
>>> test_q2()
-----------------------------------------
Testing Q2: SDES Configuration

SDES Parameters:
['encoding_type', 'block_size', 'key_size', 'rounds', 'p', 'q']

An empty configuration file is created
Get current configuration: []
Get value of parameter block_size:
Get value of parameter SDES_version: Error(get_SDES_value): invalid parameter
config_SDES('block_size',12): Error(config_SDES): invalid value  False
config_SDES('block_bits',12): Error(cofig_SDES): invalid parameter  False

config_SDES('block_size','12'):   True
config_SDES('key_size','9'):   True
config_SDES('q','503'):   True
Get current configuration: [['block_size', '12'], ['key_size', '9'], ['q', '503']]
get_SDES_value('key_size'):  9
get_SDES_value('p'):
Update value of q to 403
Get current configuration: [['block_size', '12'], ['key_size', '9'], ['q', '403']]

-----------------------------------------
```

Q3: SDES Key Generation (2 pts)

This tasks involves writing functions for generating the keys and subkeys for SDES. This involves writing three functions:

```
1-blum(p,q,m)
2-generate_key_SDES()
3-get_subKey(key,i)
```

The first function is an implementation of the Blum Blum Shub algorithm. Three values are passed, p, q and m, where m is the number of bits for the output random bit stream.

The function validates inputs by insuring that m is a positive integer and both p and q are positive integers congruent to 3 mod 4.

The seed is computed as the n^{th} prime number, where n = p*q. The prime number can be obtained from the primeFile which contains the first million primes. If the seed is not relatively prime with n, then the function should pick the next prime number and continue until a valid number is found.

Below is the function definition:

```
#-------------------------------------------------------------
# Parameters:   p (int)
#               q (int)
#               m (int): number of bits
# Return:       bitStream (str)
# Description:  Uses Blum Blum Shub Random Generation Algorithm to generates
#               a random stream of bits of size m
#               The seed is the nth prime number, where n = p*q
#               If the nth prime number is not relatively prime with n,
#               the next prime number is selected until a valid one is found
#               The prime numbers are read from the file primeFile
#               (starting n=1)
# Error:        If number of bits is not a positive integer -->
#               print('Error(blum): Invalid value of m',end =''), return ''
#               If p or q is not an integer that is congruent to 3 mod 4:
#              print('Error(blum): Invalid values of p,q',end =''), return ''
#-------------------------------------------------------------
def blum(p,q,m):
    # your code here
    return bitStream
```

The second function generates a key for the SDES scheme. First the function checks if the configuration file contains a value for the key_size parameter. If there is no such value, the function prints an error and returns an empty string. Next the function checks for values of p and q in the configuration file. If no such values are present the default value of p = 383 and q = 503 are to be used. In that scenario, the configuration file should be updated to reflect these two values. Finally, the function invokes the Blum Blum Shub Algorithm to randomly generate the key based on the proper values of p,q and m (key_size). Below is the function definition:

```
#-------------------------------------------------------------
# Parameters:   None
# Return:       key (str)
# Description:  Generates an SDES key based on preconfigured values
#               The key size is fetched from the SDES configuration
#               If no key size is available, an error message is printed
#             Also, the values of p and q are fetched as per SDES configuration
```

```
#               If no values are found, the default values p = 383 and q = 503
                are used. The configuration file should be updated.
#               The function calls the blum function to generate the key
# Error:        if key size is not defined -->
#        print('Error(generate_key_SDES):Unknown Key Size',end=''), return ''
#-------------------------------------------------------------
def generate_key_SDES():
    # your code here
    return key
```

The last function generates an SDES subkey for the i^{th} round in the Feistel Network. This key is one bit shorter than the original key, and is generated by applying a circular left shift on the key for i items, then stripping the least significant bit. Note that that value of i should be a positive integer. Therefore, i = 1 corresponds to the original key without the least significant bit. Below is the function definition:

```
#-------------------------------------------------------------
# Parameters:   key (str)
#               i (int)
# Return:       subkey (str)
# Description:  Generates a subkey for the ith round in SDES
#               The sub-key is one character shorter than original key size
#               Sub-key is generated by circular shift of key with value 1,
#               where i=1 means no shift
#               The least significant bit is dropped after the shift
# Errors:      if key is not a valid binary number or its length does not match
#              key_size: --> print('Error(get_subKey): Invalid key',end=''),
#                return ‘’
#               if i is not a positive integer:
#                   print('Error(get_subKey): invalid i',end=''), return ''
#-------------------------------------------------------------
def get_subKey(key,i):
    # your code here
    return subkey
```

Testing the three functions will give:

```
>>> test_q3()
-----------------------------------------
Testing Q3: Key Generation

Testing Blum:
blum(383,503,8):  01110011
blum(11,19,4):  1110
blum(383,503,0): Error(blum): Invalid value of m
blum(383,"503",1): Error(blum): Invalid values of p,q
blum(384,503,1): Error(blum): Invalid values of p,q

Testing generate_key_SDES:
key_size = '', p = '', q=''
generate_key_SDES(): Error(generate_key_SDES):Unknown Key Size
key_size = 9, p = '', q=''
generate_key_SDES(): 011100111
key_size = 9, p = 19, q = ''
generate_key_SDES(): 110111000

Testing get_subKey:
Key =  010011001
Subkey 0 = Error(get_subKey): invalid i
Subkey 1 =  01001100
Subkey 2 =  10011001
Subkey 3 =  00110010
Key =  308
Subkey 1 = Error(get_subKey): Invalid key

-----------------------------------------
```

Q4: Feistel Network
(2 pts)

This tasks is the implementation of the Feistel Network/Cipher. This involves writing five functions:

1-expand(R)
2-sbox1(R)
3-sbox2(R)
4-F(Ri,ki)
5-feistel(bi,ki)

The expand function operates on blocks of binary numbers which are of even size of at least 6 bits. The output is a block that is 2 bits more in size than the input block. The function swaps and repeats the two middle bits, while keeping the other bits unchanged. For example if

$R = [R_1R_2R_3R_4R_5R_6] \rightarrow [R_1R_2R_4R_3R_4R_3R_5R_6]$

$R = [R_1R_2R_3R_4R_5R_6R_7R_8] \rightarrow [R_1R_2R_3R_5R_4R_5R_4R_6R_7\ R_8]$

Below is the function definition:

```
#-------------------------------------------------------------
# Parameters:   R (str): binary number of size (block_size/2)
# Return:       R_exp (str): expanded binary
# Description:  Expand the input binary number by adding two digits
#               The input binary number should be an even number >= 6
#               Expansion works as the following:
#               If the index of the two middle elements is i and i+1
#               From indices 0 up to i-1: same order
#               middle becomes: R(i+1)R(i)R(i+1)R(i)
#               From incides R(i+2) to the end: same order
# Error:        if R not a valid binary number or if it has an odd length
#               or is of length smaller than 6
#                   print('Error(expand): invalid input'), return ''
#-------------------------------------------------------------
def expand(R):
   # your code here
   # return r_exp
```

Both sbox1 and sbox2 functions work in a similar manner, but the contents of each file is different. Description here is presented for sbox1.

The input is a block of even number of bits called ‘R’. The R size should be one bit extra than quarter of the value of block_size defined in the configuration file. If the configuration file contains no value for block_size, then an error message should be printed and an empty string is returned.

The function then access the sbox1File and retrieves the structure relevant to the R size. For instance if R size is 4, then the Sbox should fetch the line that starts with 5:xxxx in the sbox1 file.

One way to generate the output is to construct a 2D list from the retrieved sbox1 value from file, such that number of rows is 2. The output will be determined by the value of R, such that the row number correspond to the most significant bit of R, and the remaining bits correspond to the column number. Below is the function definition for sbox1:

```
#-------------------------------------------------------------
# Parameters:   R (str): binary number of size (block_size//4)
# Return:       R_exp (str): expanded binary
# Description:  Validates that R is of size block_size//4 + 1
#               Retrieves relevant structure of sbox1 from sbox1File
#               Most significant bit of R is row number,
#               other bits are column number
# Errors:       if undefined block_size:
#                print('Error(sbox1): undefined block size',end=''), return ''
#               if invalid R:
#                 print('Error(sbox1): invalid input',end=''),return ''
#               if no sbox1 structure exist:
#                 print('Error(sbox1): undefined sbox1',end=''),return ''
#-------------------------------------------------------------
def sbox1(R):
    # your code here
    return R_sbox1
```

The F function, receives an R block of data and a subkey. The function performs the following:

1- Pass the R block to the expander function
2- XOR the output of [1] with ki
3- Divide the output of [2] into two sub-blocks
4- Pass the most significant bits of [3] to the sbox1 and the least significant bits to sbox2
5- Concatenate the outputs of [4] as [sbox1][sbox2]

Below is the function definition:

```
#-------------------------------------------------------------
# Parameters:   Ri (str): block of binary numbers
#               ki (str): binary number representing subkey
# Return:       Ri2 (str): block of binary numbers
```

```
# Description:  Performs the following five tasks:
#               1- Pass the Ri block to the expander function
#               2- Xor the output of [1] with ki
#               3- Divide the output of [2] into two equal sub-blocks
#               4- Pass the most significant bits of [3] to Sbox1
#                  and least significant bits to sbox2
#               5- Conactenate the output of [4] as [sbox1][sbox2]
# Error:        if ki is an invalid binary number:
#                   print('Error(F): invalid key',end=''), return ''
#               if invalid Ri:
#                   print('Error(F): invalid input',end=''),return ''
#-------------------------------------------------------------
def F(Ri,ki):
   # your code here
   return Ri2
```

The last function applies one round of Fiestel Cipher. This is represented as:

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus F(R_{i-1}, k_i)$$

The function needs to validate that the passed block and key have similar values to those available at the configuration file.

Below is the function definition:

```
#-------------------------------------------------------------
# Parameters:   bi (str): block of binary numbers
#               ki (str): binary number representing subkey
# Return:       bi2 (str): block of binary numbers
# Description:  Applies Fiestel Cipher on a block of binary numbers
#               L(current) = R(previous)
#               R(current) = L(previous)xor F(R(previous), subkey)
# Error:        if ki is an invalid binary number or of invalid size
#                   print('Error(feistel): Invalid key',end=''), return ''
#               if invalid Ri:
#                   print('Error(feistel): Invalid block',end=''),return ''
#-------------------------------------------------------------
def feistel(bi,ki):
   # your code here
    return bi2
```

Running the testing function will give:

```
----------------------------------------
Testing Q3: Key Generation

Testing Expand:
expand(011001) = 01010101
expand(00001111) = 0001010111
expand(0011) = Error(expand): invalid input
expand(0000111) = Error(expand): invalid input

Testing Sbox1:
Sbox1(0110) = 111
Sbox1(1111) = 011
Sbox1(11110) = Error(sbox1): invalid input
Sbox1(00000) = 1011

Testing Sbox2:
Sbox2(0110) = 111
Sbox2(1111) = 100
Sbox2(11110) = Error(sbox2): invalid input
Sbox1(00000) = 0101

Testing F function:
F(100110,01100101) = 000100
F(10011,01100101) = Error(F): invalid input
F(100110,0110010) = Error(F): invalid key
```

```
Testing feistel:
feistel(011100100110,01100101) = 100110011000
feistel(01110010011,01100101) = Error(feistel): Invalid block
feistel(011100100110,0110010) = Error(feistel): Invalid key

----------------------------------------
```

Q5: SDES Encryption and Decryption (2 pts)

Now, that all of the components of the SDES are implemented the encryption and decryption functions are to be implemented.

Both functions start by breaking the input into blocks of two characters. A padding of Q is added if necessary. Each block is passed to the Feistel Cipher

as many as the value defined in the parameter ‘rounds’. At the end of the encryption of each block, the two halves of the output are swapped.

The decryption is similar to the encryption function with two exceptions. The first is that the input is swapped instead of the output. Second, the keys are generated in the reverse order.

Although it would be better to design your encryption and decryption function to be as generic as possible, for the purposes of this assignment, it is enough to make it work for the B6Coding.

The encryption function validates the input plaintext through ensuring that it is a non-empty string. It also validates the configuration, by ensuring that the following parameters are configured in the file: coding_type, block_size, key_size and rounds. If one of these is undefined, an error message is printed and the function is aborted.

If the function does not receive a key (i.e. the key is an empty string), then it fetches the values of p,q and key_size from the configuration file and generates the key, using the functions created in Q2. If one of these parameters is undefined, it aborts and returns an empty string. However, if a key is passed, then the function validates it against the key_size.

The encryption function encrypts only characters that are defined in the encoding scheme. The defined characters are grouped in blocks. In the specific example of B6 encoding, every two characters are grouped together. Get use of the utility functions that mark, remove and insert back the undefined characters. Also, remember to pad your plaintext with ‘Q’ if necessary.

Each block should be translated into a binary steam. The stream will be divided based on the value of block_size and passed to the Feistel Cipher as many as the value of the value parameter. Also, remember to swap the two halves at the end of the last round.

When the above is complete, translate the binary stream (this is the cipher bit stream) back into English through the proper encoding scheme.

The decryption process is similar to encryption with the exception that it works in the other way round. This is specifically important for the order of subkeys used.

Below is the output of running the testing functions:

```
>>> test_q5()
-------------------------------------------
Testing Q5: SDES Encryption & Decryption

Testing for invalid input:
e_SDES(,): Error(e_SDES): Invalid input
d_SDES(,): Error(d_SDES): Invalid input
e_SDES(123,): Error(e_SDES): Invalid input
d_SDES(123,): Error(d_SDES): Invalid input

Testing when configuration file is empty:
e_SDES(abc,1101): Error(e_SDES): Invalid configuration
d_SDES(abc,1101): Error(d_SDES): Invalid configuration

Setting the Confgiruation File
Configuration =  [['rounds', '2'], ['key_size', '9'], ['block_size', '12'],
['encoding_type', 'B6']]
e_SDES(abc,1101): Error(e_SDES): Invalid key
d_SDES(abc,1101): Error(d_SDES): Invalid key

Testing one block encryption/decryption:
e_SDES(yR,111000111) = da
d_SDES(da,111000111) = yR
```

```
Testing one block encryption/decryption with undefined characters:
e_SDES((y--R),111000111) = (d--a)
d_SDES((d--a),111000111) = (y--R)

Testing multiple blocks encryption/decryption:
e_SDES(DES is a legacy Cipher,111000111) = hB2AmSQDvS4X6SkMwmkLZp
d_SDES(hB2AmSQDvS4X6SkMwmkLZp,111000111) = DES is a legacy Cipher

Testing padding and empty key:
e_SDES(DES,) = nOq9
d_SDES(nOq9,) = DES

Wha happens when all blocks are the same?
e_SDES(AAAAAAAA,111000111) = MXMXMXMX
d_SDES(MXMXMXMX,111000111) = AAAAAAAA

-------------------------------------------
```