Simple Substitution Ciphers

Introduction

In this exercise, we will complete several pieces of Python programs to encode and decode messages by using simple substitution ciphers.

Objectives

The purpose of this assignment is to help you:

• Refresh knowledge on string, list and dictionary.

• Learn to write functions, and to use `join` method in Python.

• Learn list and dictionary comprehension.

• Learn to follow the additional requirements.

Background

One simple substitution cipher

In this project, we will write some code to encode and decode messages. There is a simple kind of coding scheme called substitution cipher in which every letter of the alphabet is mapped to a different letter. A simple case of this might be described as follows:

```
alphabet: ABCDEFGHIJKLMNOPQRSTUVWXYZ
codestring: JKLMNWXCDPQRSTUVAEFOBGHIZY
```

Each letter in the top line (aka. in the alphabet) will get changed into the corresponding letter in the bottom line (aka. in the codestring). For example, the string `HELLO` is encoded as `CNRRU` using the code above. The string `GDLOUEZ` is decoded as `VICTORY`. We are calling the string to be encoded (e.g. `HELLO`) and the decoded result (e.g. `VICTORY`) plaintext. And we are calling the encoded string (e.g. `CNRRU`) and the string to be decoded (e.g. `GDLOUEZ`) ciphertext.

The alphabet rarely changes and people usually use different codestring for different cipher

applications. In this assignment, if not specified, the alphabet will always be solid as `ABCDEFGHIJKLMNOPQRSTUVWXYZ` and the alphabet and codestring are with the same length.

Assignment

When you are reading this assignment, you must have already downloaded the skeleton zip file. In the zip file, you can find a skeleton code, named cipher.py. All you need to do is to complete the skeleton code based on the instructions and submit it to the Mimir system.

I. Substitution cipher with string operation (Done in the lab)

Before we are going to work on the skeleton code, we will first try some simple implementations of substitution cipher. It is not too hard to decode such a code if you know the ciphertext for the whole alphabet. Here is a little code that prints a decoded output.

```
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
codestring = "BCDEFGHIJKLMNOPQRSTUVWXYZA"
ciphertext = "IFMMPXPSME"
for char in ciphertext:
    print(alphabet[codestring.index(char)], end = "")
```

Output

```
HELLOWORLD
```

If you wanted to produce a plaintext but not print the result, you might try something like the following:

```
plaintext = ""
for char in ciphertext:
    plaintext = plaintext + alphabet[codestring.index(char)]
```

Now that you know how the decode function for string operation works, write the encode function.

```
encode_string(codestring, plaintext)
decode_string(codestring, ciphertext)
```

Note: You must complete both the encode and the decode functions.

II. Substitution cipher using lists

Part 1

Create two functions `create_elist(codestring)` and `create_dlist(codestring)` that create and return the lists (`e_list` and `d_list` respectively), which you will use as lookup while encoding and decoding. Consider the following example.

```
alphabet = "ABCDE"
codestring = "CDEAB"
```

The two lists will be as follows.

```
e_list = ['C', 'D', 'E', 'A', 'B']
d_list = ['D', 'E', 'A', 'B', 'C']
```

Here, `'A'` encodes to `'C'`, and `'C'` decodes to `'A'`. So `e_list[0] = 'C'`, where 0 is the index of `'A'` in `alphabet`. And `d_list[2] = 'A'`, where 2 is the index of `'C'` in `alphabet`.

Similarly, `'B'` encodes to `'D'`, and `'D'` decodes to `'B'`. So `e_list[1] = 'D'` where 1 is the index of `'B'` in `alphabet`. And `d_list[3] = 'B'` where 3 is the index of `'D'` in `alphabet`.

To understand this better, try to do the above for the remaining characters on paper.

It is very common in Python to produce a list by iterating over a different list. As a result, there is a special syntax for it, called list comprehension. Here is an example for list comprehension.

Consider the following code to generate a list that holds the squares of the elements in `num`.

```
num = [1, 2, 3]
squares = []
for i in num:
    squares.append(i**2)
```

The same code using list comprehension:

```
squares = [i**2 for i in num]
```

Use this concept in `create_elist` and `create_dlist`.

Part 2

Now, create two functions `encode_list(e_list, plaintext)` and `decode_list(d_list, ciphertext)`. Analogously to the first section, these functions will create a list and append to it. This is less time consuming. Then, in order to get a string at the end, we can use the join method. Technically, join is a string method, so we can call it on the string when we want to combine the individual elements of the list. Here is an example for the join method.

```
list1 = ['H', 'E', 'L', 'L', 'O']
string1 = "".join(list1)
```

(The empty string `""` act as the delimiter.)

Output

```
HELLO
```

Using `e_list` and `d_list` as lookup write the functions `encode_list` and `decode_list` functions. These functions should return the ciphertext and plaintext as strings(use the join method).

Note: If we start with any string S, the following code MUST return S again.

```
##this will work for any codestring whose
##length is the same as that of the alphabet
e_list = create_elist(codestring)
d_list = create_dlist(codestring)

# for any string S, decode_list should return the same thing
cipher = encode_list(e_list, S)
S_new = decode_list(d_list, cipher)
```

III. Substitution cipher with dictionary operation

Lists and strings gave us a way to map between encoded and decoded letters. An even better

way to map encoded letters to their corresponding decoded letters is to use a dictionary. (A dictionary is also known as a mapping.)

Part 1

Write two functions `create_edict(codestring)` and `create_ddict(codestring)` that create and return the lists (e_dict and d_dict respectively), which you will use as lookup while encoding and decoding. Again, consider the following example.

```
alphabet = "ABCDE"
codestring = "CDEAB"
```

The two dictionaries will be as follows.

```
e_dict = {'A':'C', 'B':'D', 'C':'E', 'D':'A', 'E':'B'}
d_dict = {'C':'A', 'D':'B', 'E':'C', 'A':'D', 'B':'E'}
```

In `e_dict`, each character of the alphabet acts as the key and the corresponding character of the codestring acts as the corresponding value. `d_dict` is the other way round. Later you will use these two dictionaries to perform encoding and decoding. Here is an example to create a dictionary.

```
keys = ['Name', 'Age', 'Country']
values = ['ABC', '20', 'USA']
dict1 = {}
for i in range(3):
    dict1[keys[i]] = values[i]
```

This will create the following dictionary.

```
dict1 = {'Name':'ABC', 'Age': 20, 'Country':'USA'}
```

In case you were wondering if there is dictionary comprehension in the same way that there is list comprehension, there is! So, we could create the `code` dictionary by first turning the `alphabet` and `codestring` into a list of pairs (tuples) and doing a dictionary comprehension. Here is the same dictionary as above, created using dictionary comprehension.

```
keys = ['Name', 'Age', 'Country']
values = ['ABC', 20, 'USA']
dict1 = {k:v for (k,v) in zip(keys, values)}
```

Use this concept in `create_edict` and `create_ddict`.

Note: Please refer to the following link for information on `zip()`:
https://docs.python.org/3/library/functions.html#zip

Part 2

Now, create two functions, write `encode_dictionary(e_dict, plaintext)` and `decode_dictionary(d_dict, ciphertext)`. These will work very much as in section II, but except that the lookup will be done with dictionaries rather than lists.

Note: If we start with any string S, the following code MUST return S again.

```
## this will work for any codestring
## whose length is the same as that of the alphabet
e_list = create_edict(codestring)
d_list = create_ddict(codestring)

# for any string S, decode_list should return the same thing
cipher = encode_dictionary(e_list, S)
S_new = decode_dictionary(d_list, cipher)
```

VI. Additional requirements

Before you are done, you MUST adapt our functions so that they automatically convert `plaintext`, `ciphertext`, and `codestring` to uppercase before using them. We can use the `str.upper()` method for that. This method returns an uppercase version of the string. For example, `'Hello'.upper()` returns `HELLO`.

The second thing we need to consider is that the encode and decode functions MUST handle the case when the plaintext has characters that are not in the alphabet, or the cypher has characters that are not in the codestring. These characters should be left as-is (not encoded or decoded). This will mean that before looking up a character, you need to check if it is in your list or dictionary.

However, if there are spaces in the plaintext, we should change them to dashes (-) in the ciphertext, and vice versa, rather than leaving them as-is.

After implementing the requirements above, if we have a codestring `JMBCYEKLFDGUVWHINXRTOSPZQA` , then given plaintext `Ab3c, De1::6` , the encoded ciphertext should be `JM3B,-CY1::6` . With the same codestring, if ciphertext `A-p4s#%!`` is given, the decoded plaintext should be` Z W4V#%!`.

Submit your work to Mimir

Submit your code to Mimir after you complete your code. Mimir will automatically grade your submission based on different unit test cases (with different codestring, plaintext and ciphertext). You can submit your code to Mimir any number of times to refresh your existing score before the submission deadline.