Lab02: Simple Substitution Ciphers
CSE2050 Fall 2017
Instructor: Jeffrey Meunier & Wei Wei
TA: Jenny Blessing, Param Bidja, Yamuna Rajan & Zigeng Wang

1. Introduction

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

2. Objectives

The purpose of this assignment is to help you:

- Refresh knowledge on string, list and dictionary.
- Refresh knowledge on object-oriented design.
- Do a little bit study on list comprehension and dictionary comprehension.
- Learn to use zip function in Python.

***Note**: Before you start, if you are not familiar with string, list, dictionary, loops or basic object-oriented design in Python, you are recommended to review the sample codes we covered in lecture first.*

3. Background

3.1. 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: ***A B C D E F G H I J K L M N O P Q R S T U V W X Y Z***

codestring: ***J K L M N W X C D P Q R S T U V A E F O B G H I Z Y***

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 call the encoded string (e.g. “CNRRU”) and the string to be decoded (e.g. “GDLOUEZ”) **ciphertext**.

The **alphabet** rarely changes and people usually change **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.

4. 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.

4.1. Substitution cipher with string operation

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 = "IFMMMPXPSME"
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)]
```

4.2. Substitution cipher with list operation

As we can see in the string operation above, concatenating strings creates a whole new string each time, which is pretty time-consuming.

Appending to a string is slow, but appending to a list is faster. So, instead, we could do this by creating a list and appending each letter to the list. Then, in order to get a string at the end, we can use the ***join*** function. 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 like the code as the following:

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

It is a *very* common operation 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 the above code again, using list comprehension.

```
# List Comprehension is the right tool in this case.
plaintextlist = [alphabet[codestring.index(char)] for char in ciphertext]
plaintext = "".join(plaintextlist)
```

We can further package all the things above into a function as the following (together with some test cases):

```
def decode(codestring, ciphertext):
    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    plaintextlist = [alphabet[codestring.index(char)] for char in ciphertext
    return "".join(plaintextlist)

code1 = "BCDEFGHIJKLMNOPQRSTUVWXYZA"
code2 = "CDEFGHIJKLMNOPQRSTUVWXYZAB"

print(decode(code1, "IFMMPXPSME"))
print(decode(code1, "TFDSFUTFDSFU"))
print(decode(code2, "FKHHGTGPVEQFG"))
```

```
# Output
HELLOWORLD
SECRETSECRET
DIFFERENTCODE
```

4.3. Substitution cipher with dictionary operation

One slightly annoying thing about the code above is that it requires us to find the index of each character in the ciphertext as it appears in the alphabet string. This isn't so bad, but it does require searching through the whole string. That is, it's about 26 times slower than a normal list access. Imagine if we also had lowercase letters and punctuation. It could be 100 times slower. Again, for tiny problems you can't see the difference, but as soon as you need to decode millions of messages, the difference between 5 minutes and 500 minutes, is a lot. A better way to *map* encoded letters to their corresponding decoded letters is to use a dictionary. (A dictionary is also known as a *mapping*.)

We can create a dictionary from the code string as follows.

```
codestring = "BCDEFGHIJKLMNOPQRSTUVWXYZA"
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

```
code = {}
inverse = {}
for i in range(26):
   code[alphabet[i]] = codestring[i]
   inverse[codestring[i]] = alphabet[i]
```

we can decode a ciphertext as follows.

```
ciphertext = "IFMMPXPSME"
plaintext = "".join([inverse[c] for c in ciphertext])
```

When you are comfortable with list comprehensions, you will find this version very satisfying because it does pretty much exactly what it says: make a list of characters where each character is the inverse of the next character in the ciphertext, then join them back up. Was that too fast? Break it into pieces.

```
ciphertext = "IFMMPXPSME"
# Use list comprehension to convert the letters to a decoded list
listofplaintext = [inverse[c] for c in ciphertext]
# Join the list of plaintext letters into a string.
plaintext = "".join(listofplaintext)
```

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. There is a convenient function called ***zip*** that does this for us. So, the following code could create a code dictionary.

```
code = {b:a for (a,b) in zip(codestring, alphabet)}
inverse = {a:b for (a,b) in zip(alphabet, codestring)}
```

If this is terribly confusing, try making two lists in a python shell and ***zip*** them. What is the result? Play around with some small examples. Imagine an alphabet of only 4 letters perhaps to keep things short. You can also try searching the Internet to find some helpful examples.

4.4. Packaging things into a class

Now, it is time to open the skeleton code cipher.py. We could start with a mostly empty class called Cipher stored in the file as the following.

```
class Cipher:
   def __init__(self, codestring):
      pass
```

```
def encode(self, plaintext):
    pass

def decode(self, ciphertext):
    pass
```

In the code above, we are given a class called Cipher. Class Cipher comes with a constructor, but currently it doesn't do anything. As we know, the methods of a class all accept a variable called **self** as their first parameter. In this case, **self** can be used to access attributes or other methods. In our case, we will make an attribute to store the code. We will make another attribute to store the inverse of the code.

Now, add your code in the constructor for the Cipher class that takes a **codestring** and calculates both the **code** and its **inverse** in two dictionaries, like what we did in Section4.3. The **code** and its **inverse** should be attributes (member variables) that will be used in methods of the Cipher class.

After that, we need to complete two other methods, **encode** and **decode**. The **encode** method takes a plaintext as input and returns the corresponding ciphertext, while the **decode** method takes a ciphertext as input and returns the corresponding plaintext.

4.5. Some other concerns

Besides the basic implementations in Section 4.4, there are two more things we need to do.

The first thing is to adapt our code so that it automatically converts plaintext, ciphertext, and codestring to uppercase, where we can use the **str.upper()** method. 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** methods should leave all punctuation marks in place. This will mean that you should check if the letter is in the **code** or its **inverse** and leave the letter if not. Checking if a key **k** is in a dictionary **D** can be done by writing **if k in D**. Moreover, if there are spaces in the plaintext, we should change the spaces into dashes (-) in the ciphertext, and vice versa. We assume, there will be no dashes in plaintext, and no spaces in ciphertext.

After implementing the two things above, if we have a certain codestring "JMBCYEKLFDGUVWHINXRTOSPZQA", given plaintext "Ab3c, De1::6", the encoded ciphertext should be "JM3B,---CY1::6". With the same codestring, if ciphertext "--Ap4s#$!" is given, the decoded plaintext should be " ZW4V#$!".

5. Submit your work to Mimir

Submit your code to Mimir after you complete your code. The 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 up to **30 times** to refresh your existing score before the submission deadline.

6. Due date

This lab assignment is worth **2 points** in the final grade. It will be due by **11:59pm on Wednesday, Sep 20th 2017**. A penalty of **10% per day** will be deducted from your grade, starting at 12:00am.