Simple Substitution Ciphers

In this project, we will write some code to crack codes. There is a simple kind of code called a **substitution cipher** in which every letter of the alphabet is mapped to a different letter. You will write some code to encode and decode messages. The details of your task are included in this document, so you will really have to read it.
A simple case of this might be described as follows:

```
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
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 (called the *plaintext*) will get changed into the corresponding letter in the bottom line (called the *ciphertext*). For example the string `"HELLO"` is **encoded** as `"CNRRU"` using the code above. The string `"GDLOUEZ"` is **decoded** as `"VICTORY"` .

It is not too hard to decode such a code if you know the ciphertext for the whole alphabet. We'll call this the **codestring**. 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
```

Lesson: Appending to a string is slow. Appending to a list is faster.

If you wanted to produce a string and not just print the output, you might try something like the following.

```python
# Don't do this!
# Concatenating strings creates a whole new string each time.
plaintext = ""
for char in ciphertext:
    plaintext = plaintext + alphabet[codestring.index(char)]
```

You could do this instead by using a list and appending to the list. Then, to get a string at the end, you use the `join` function. Technically, `join` is a string method, so you call it on the the string you want to separate the individual elements of the list.

```python
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.

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

Packaging this into a function

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

code1 = "BCDEFGHIJKLMNOPQRSTUVWXYZA"
code2 = "CDEFGHIJKLMNOPQRSTUVWXYZAB"

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

```
# Output
HELLOWORLD
SECRETSECRET
DIFFERENTCODE
```

Storing a code as a dictionary

One slightly annoying thing about the code above is that it requires us to find the index of each character in the cyphertext 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]
```

Now, 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.

Packaging this into a class

We could start with a mostly empty class called `Cipher` stored in a file called *cipher.py*. Here is its contents.

```
# contents of cipher.py
class Cipher:
    def __init__(self):
        pass
```

In the code above, we created a new class. We also added a constructor, but it doesn't do anything. 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. We will add two methods, `encode` and `decode`.

Testing our code

Next, we will write some code to test it. We'll use the `unittest` package. Here is a sample test file.

```python
#contents of testcipher.py
import unittest
from cipher import Cipher

class TestCipher(unittest.TestCase):
    def testcreate(self):
        pass

if __name__ == '__main__':
    unittest.main()
```

The `unittest.main()` function finds all the classes that inherit from `unittest.Testcase` and runs all the methods that start with the word `test`. The line `if __name__ == '__main__':` checks that the test file itself is being run, not just imported.

Your mission

1. Write a constructor for the Cipher class that takes a codestring and stores both the code and its inverse in two dictionaries.
2. Write an `encode` method for the `Cipher` class that takes a plaintext as input and returns the corresponding ciphertext.
3. Write a `decode` method for the `Cipher` class that takes a ciphertext as input and returns the corresponding plaintext.

Some other concerns

4. Adapt your code so that it automatically converts plaintext, ciphertext, and codestrings to uppercase. Use the `str.upper()` method. This method returns an uppercase version of the string so, for example, `'Hello'.upper() == 'HELLO'`.
5. The `encode` and `decode` methods should leave all spaces and punctuation in place. This will mean that you should check if the letter is in the code and leave it if not. Checking if a key `k` is in a dictionary `D` can be done by writing `if k in D:`.