June 9th, Substitution Ciphers

Never trust a computer you can't throw out a window.–Steve Wozniak

We are going to implement substitution ciphers and then make some tools to help us break them.

First we need to think about a way to represent substitution ciphers. Here is one way. A substitution cipher can be represented as a string of 26 distinct letters. The first letter can represent where `a` is sent, the second where `b` is sent, and so on.

For instance `"qwertyuiopasdfghjklzxcvnm"` would mean that $\texttt{a} \mapsto q, \texttt{b} \mapsto w, \ldots, \texttt{z} \mapsto m$.

Let's make a function in Sage which encrypts things. It will take in a string `key` which is a key like the one above. It will also take in a string `plaintext`. Here's the logic.

- Take the letter of plaintext you want to encrypt (we'll run this in a loop).
- Turn it into a number between 0 and 25 using the `ord` function (and a shift). Call this number `c`.
- Look up the corresponding letter in the `key`, ie `key[c]` should be the new letter.
- Store this new letter in your output string.

Here's how my function looked. You have to fill in the ... yourself with several lines of code.

```
def substitutionEncrypt(plaintext, key):
    outString = ''
    for i in plaintext:
        ...
    return outString

print substitutionEncrypt("abc","qwertyuiopasdfghjklzxcvbnm")
```

which output `qwe` as expected.

1. Make your own function that does the same thing.

Now we want to write a function `substitutionDecrypt(ciphertext,key)` that decrypts a message that has been encoded via a substitution cipher with keyword `key`. Remember that to encrypt a letter we change it to the letter in the corresponding location of the keyword (for example, `a` encrypts to the first letter of the keyword, `z` encrypts to the last letter of the keyword). This means that to decrypt a letter we first find it in the keyword and then change it to the letter corresponding to that location (for example, since `r` is the forth letter in the above keyword, `r` decrypts to `d`).

Our Strategy to decrypt is as follows:

- Take the letter of ciphertext you want to decrypt.
- Find where it is in the key string (i.e., what position). Call that `loc`.
- The decrypted letter will be the letter corresponding to the number `loc`.

Thankfully, Sage has built in functions that tell you where a character appears in a string. Notice that if you run

```
s = "abc"
loc = s.find("b")
print loc
```

then you will have printed where "b" appears in the text "abc"; in position number 1 in this case. See what happens when you ask for the location of a character that is not in the string.

2. Make a function that decrypts things made with the substitution cipher.

Breaking the substitution cipher

Figuring out the keyword used to encode a substitution cipher becomes much easier when you realize that some parts of the keyword are easier to figure out than others because some letters appear in common English more often than others. For example, suppose I have Sage count how many times each letter appears in the encoded message and it tells me that the most common letters are `q, x, l, n,`. I know that the most common letters in common English are `e, t, a, o` so it makes sense to guess maybe q is e, x is t, l is a and n is o. This may be wrong, but it gives a place to start guessing.

It is also useful to examine digraphs[1] and use the fact that the most digraphs are `th, he, in, er, an, ....`

Analyzing the frequency of individual letters as well as digraphs may allow you to make very good guesses as to what some of the letters of the keyword are. You can use this information to partially decode the message, and looking at the partially decoded message may suggest what other letters should decode to.

The outline of the strategy is as follows:

- Analyze the frequency of individual letters and digraphs to make initial guesses about some letters of the keyword.
- Decrypt part of the message with this partial keyword.
- Stare at partially decrypted message and try to guess other letters of the keyword.

First, let's make a function that let's us put in part of a key and spits out some partial plaintext. Here's how the function I made worked.

- I take in the ciphertext as a string `ciphertext`.
- I also receive a string with 26 characters in it. If I think `e` was converted to another letter, I put that other letter in the in the 5th spot in the string (the computer calls the 5th spot the 4th spot). The letters I don't want to do anything to, I put a pound sign `'#'` there.

For instance. If I think the key that the person used to make the substitution cipher turned `e` into `c`, then I might have this string be

`####c####################`

Ok, now make a function which partially decrypts based on such a string. I find it useful for my decrypted letters to be uppercase while my encrypted letters to be lower case. For example, if I have the ciphertext `abc` and I use the partial key above, then my function would output `abE`. This makes it easy to to try out parts of keys and see if I'm getting close to decrypting. My function

`def substitutionPartialDecrypt(ciphertext, key):`

looks almost exactly like my regular decryption function, except I have some additional logic in the middle so that it only changes characters I am guessing about.

```
...
if (loc >= 0):
    outString = outString + chr(ord('A')+loc)
else:
    outString = outString + i
...
```

4. Make your own function that partially decrypts based on a partial key.

5. Next make a function that lists how frequently each letter occurred. You already made a *more* complicated version of this function when we broke Vigenére.

[1] Pairs of two letters.

Now to write a function that determines the frequency of digraphs. First, let us remember some useful operations for strings. For any string `s` you can always take a substring `s[a:b]` where `a` is the position of the first character to take and `b` is the position where you stop taking characters. For example, try running

```
s = "abcde"
print s[1:3]
```

which should return `"bc"`.

Also experiment with the function `count` which tells you how many times a substring occurs.

```
s = "abcdeab"
print s.count("ab")
print s.count("bc")
```

My function is rather naive, it goes through my string, looking at each pair of consecutive letters and counts how many times they occur. Some things are listed more than once. You can make your function smarter if you want (or only print out digraphs that occur at least 5 or 10 times?). My function started like this.

```
def listCommonDigraphs(ciphertext):
    for i in range(0, len(ciphertext)-1):
        ...
```

7. Make your own function. Make sure to test it out and confirm it behaves like it should.

8. Use all your functions to decrypt the ciphertext available at:

`http://www.math.utah.edu/~schwede/Camp2016/Jun9SubstitutionBreakable.txt`