Deciphering Ciphers

Since ancient times humans have felt the need to hide their communications from others. Written messages had to somehow be encrypted so that only their intended recipient could read them. (This was most important for military communications, but could also be used to protect business, religious, or personal secrets.) These encryption schemes, or *ciphers*, were usually based on replacing letters of the message in some particular way (or, less often, on reordering the letters). In this project we will write code that can encrypt or decrypt a message with three different schemes.

All the ciphers we'll be looking at in this project are part of *classical* cryptography. While we call it "classical," it covers almost all of human history and culminates in the very elaborate encryption machines used during World War II. Cryptography holds a special place in the history of computing as the need to break Nazi codes motivated the creation by the British of the first roughly-modern computers. Computers changed cryptography forever. What was impossible to break without them became trivial to break with them. But while computers rendered all previous cryptography obsolete, the theorists studying the fundamental math behind computers also came up with new, much more secure cryptography. That *modern* cryptography gives us ways not just to send secret messages but also to carry out a variety of other computational tasks even when an adversary is trying to get in the way, and that is what you would study if you take a cryptography course today.

For this project we will stick with classical cryptography, and as such we will not just write code to encrypt and decrypt messages as intended, but also code that an adversary could use to break the privacy of the encryption. The project has six parts, and each will have you write several functions to either create or break a particular encryption scheme. You will write a lot of functions in this project, but many of them will be very simple. We break up the code this way because it is good coding practice and makes writing the code easier.

Terminology

Any encryption scheme consists of two algorithms. The first is the encryption algorithm. It takes two inputs, the *plaintext*, which is the readable text we want to encrypt, and the *key,* which is some piece of secret information that we need to do the encryption. The output of the encryption algorithm is called the *ciphertext*, and it is the encrypted message, supposedly unreadable by an adversary. The second algorithm is the decryption algorithm. It takes as input a ciphertext and a key. If the key is the same key that was used to create the ciphertext, then the decryption algorithm should output the original plaintext.

We will also write algorithms for *breaking* a given encryption system. A break (or *attack*) is an algorithm that can recover the plaintext from a ciphertext *without* knowledge of the secret key.

(In practice, attacks that can learn something about the plaintext, even if they can't fully recover it, are worrying, but our attacks in this project will all fully recover the plaintext.)

Starting code

You will write almost all the code in this project yourself, but we've given you a couple basic things to start with. First, we've defined a variable `alpha` at the top of the code, equal to a string that has the English alphabet in order in all capital letters. This will be convenient when you want to know the letter at a given position of the alphabet, the position of a given letter, or whether some character is a letter.

We have also provided two basic functions for reading and writing files. Reading and writing files is best done with more elaborate techniques than these, and we will talk about it later in the course. For now, though, these functions will suffice. We will think of our messages (either plaintext or ciphertext) as being single strings. Encryption only really makes sense if you can take a message and send it to another person, and storing our encryptions in a text file will make that possible. (It will also let you use encryptions we have already prepared and included as part of the project.) `file_to_string` returns a string equal to the contents of a given file, and `string_to_file` writes a given string to a specified file. (Files will always be in the same directory as the program you are running. If you write to a file that already exists, it will erase that file and replace it with a new one.)

Finally, we have imported the `random` module, since later on you will need to use a function or two from that.

Part 1: A working Caesar cipher

Perhaps the most basic cipher is the *Caesar cipher*, also called a *shift cipher*. It is known to have been used by Julius Caesar to encrypt military communications. In a Caesar cipher, encryption works by shifting each letter forward in the alphabet a certain amount, wrapping around to the beginning if that shift would pass the end of the alphabet. The amount of the shift is the key, and we will represent it with a letter. We'll zero-order the letters, so A represents a shift of 0, B a shift of 1, C a shift of 2, and so on. For example, if one were to encrypt the message "HAPPYBIRTHDAY" with a key of D (meaning a shift of 3), we would get the following:

```
plaintext:    H A P P Y B I R T H D A Y
ciphertext:   K D S S B E L U W K G D B
```

Note that the Y shifts 3 but wraps around, going to Z, then A, then finally B. For simplicity, our messages will always consist of only capital letters. (Generally, if you remove punctuation and capitalize all letters, English speakers are still able to interpret a message reasonably easily.)

`simplify_string`

Because our ciphers will all use these simplified strings of only capital letters, we can't really work with anything until we can first simplify general text. This function should take as input a string and should output a new string that contains only the letters in the old string (removing all punctuation, spaces, and newline symbols). Those letters should also all be capitalized. The `upper` function built into Python will be useful – look for it in the documentation.

For example, `simplify_string("Happy birthday!")` returns `"HAPPYBIRTHDAY"`.

`num_to_let`

This is a simple (but very useful) function. It should take as input a single number and it should return the associated (capital) letter. Remember that we start numbering the alphabet with A numbered 0.

`num_to_let(0)` returns `"A"`.
`num_to_let(2)` returns `"C"`.
`num_to_let(24)` returns `"Y"`.

`let_to_num`

This reverses the function above. It takes a capital letter and returns the corresponding number.

`let_to_num("A")` returns `0`.
`let_to_num("C")` returns `2`.
`let_to_num("Y")` returns `24`.

`shift_char`

This function takes as input two letters. The first is a starting letter and the second is a letter indicating how much to shift by (so a D indicates a shift by 3). The output is the result of shifting the first letter by the amount specified by the second.

`shift_char("A", "D")` returns `"D"`.
`shift_char("C", "F")` returns `"H"`.
`shift_char("Y", "D")` returns `"B"`.

`caesar_enc`

This is the final encryption function. It takes as input a plaintext (a simplified string) and a key (a single capital letter). It returns the ciphertext, generated by shifting each letter in the plaintext forward by the amount specified by the key.

`caesar_enc("HAPPYBIRTHDAY", "D")` returns `"KDSSBELUWKGDB"`.

`caesar_dec`

This reverses the encryption above. It takes a ciphertext and a key, and shifts all letters *backwards* by the amount specified by the key, so that if the key generated the ciphertext, the output is the correct plaintext.

`caesar_dec("KDSSBELUWKGDB", "D")` returns `"HAPPYBIRTHDAY"`.

We have included the file `hitchhiker's.txt`, which includes an excerpt from *The Hitchhiker's Guide to the Galaxy*. Try encrypting it with your `caesar_enc` function. Then decrypt it using `caesar_dec`. It should match the original text.

Part 2: Breaking the Caesar cipher

Now that the Caesar cipher is working, it's time to break it. The Caesar cipher is not very secure at all. You could break it by hand easily. You could just try all 26 possible keys and see which one gives you something that looks like proper English. That would work with a computer too, but we'd need it to be able to recognize "proper English," which can be done but is somewhat involved. Instead we'll do something else.

We will use frequency analysis. English text doesn't contain all letters equally often. The most common letter is clearly E, with A, I, O, and T all very common as well. The simplest frequency-based attack would be to simply find the most common letter and assume that it is meant to be E, and try whatever key would decrypt that letter to E. That will work pretty often.

Frequency-based attacks seem obvious to us now, but they actually took a long time to develop. In fact, the Caesar cipher probably worked well when Julius Caesar used it. (Many of his adversaries were not even literate, and the ciphertexts might have been assumed to be an unknown language.) The discovery of frequency analysis would have to wait until al-Kindi in the 9th century. By the time the Russian army used a Caesar cipher in 1915 (because their troops had difficulty with more complicated methods), the German and Austrian cryptanalysts were easily able to decrypt their messages. Even today, there will occasionally be reports of organized crime or terrorist organizations using a Caesar cipher, with minimal effectiveness.

`letter_counts`

The first thing we need to do in order to make our frequency analysis work is the ability to count how often letters appear in a given string. `letter_counts` takes as input a string and returns counts of how many time each letter occurs, stored in a dictionary. The dictionary should always have 26 keys, one for each letter. The keys should be single-character strings of a capital letter, and the value stored should be the number of times that letter occurred in the input string. It's fine to assume that the input string is simplified (i.e., has only capital letters).

`normalize`

We really want our dictionaries to have the fraction of the string represented by each letter, not the raw counts. Normalize should take as input a dictionary of counts and should modify the dictionary (*not* return a new one) by dividing each count by the total of all the counts in the dictionary. (We'll use this again later with a dictionary with different keys, so do not assume that the keys are always the 26 letters. This function should work with any dictionary where all the values are positive integers.)

Once this is working, you can uncomment the two lines of code below it. We have included a file `twocities_full.txt` that contains the entire text of *A Tale of Two Cities*. We will use this as a large corpus of English text from which to calculate frequencies. (To truly do this correctly, you'd want to use a corpus consisting of various works of different kinds and by different authors, but this will be good enough for our needs.) The variable `english_freqs` will now give the rough frequency of each letter in true English, and we will use it for comparison. (By computing it only once and saving it we'll make our code faster, since we'll need this data many times and we don't want our program to be constantly re-computing it.)

Whenever we refer to the "frequencies" of letters in a given string, we always mean the normalized fractions, not the raw counts.

`distance`

Now, for given frequencies of letters in some text, we want to be able to measure how "far" that distribution is from the distribution of letters in standard English. We'll do that using the following formula:

$$distance = \sum_x \frac{(obs_x - exp_x)^2}{exp_x}$$

Here x represents a letter, and we're summing over all possible letters. obs_x is the frequency of x observed in the string, while exp_x is the expected frequency in standard English.

The `distance` function computes the above distance formula. It takes two inputs. The first is a set of frequencies observed in some particular piece of text, and the second is known frequencies for standard English. It should output a positive real number.

`break_caesar`

This function should attack a string encrypted with a Caesar cipher and return the original plaintext. It takes two inputs, the ciphertext and a dictionary of frequencies in standard English. Its output should be a list of two elements. The first is a single-character string equal to the key used to encrypt/decrypt the ciphertext. The second is the recovered plaintext.

To do this, you should simply try each possible key. For each one, decrypt the ciphertext to get a possible plaintext. Then measure the distance between letter frequencies in that plaintext and letter frequencies in standard English. Return whatever key gives you the lowest distance value, and the plaintext that goes with it.

`break_caesar("KDSSBELUWKGDB", english_freqs)` would return `["D", "HAPPYBIRTHDAY"]`.
(This test won't work because the message is too short for frequency analysis to be reliable, but it should give you an indication of the specifications.)

You can return to the text from `hitchhiker's.txt`, which you encrypted before. Now try breaking your ciphertext without the key, using `break_caesar`. You should be able to get the same result. We have also included a file, `mystery1.txt`, which is already encrypted with a Caesar cipher. You should now be able to break that encryption.

Part 3: A working Vigenère cipher

Caesar ciphers can be generalized to a class called *substitution ciphers.* We will get back to those in Part 5. But for hundreds of years after al-Kindi first described frequency analysis, all ciphers in use were vulnerable to it. The effort to frustrate frequency analysis eventually led to the Vigenère cipher. The Vigenère was reinvented many times, the first time by Giovan Battista Bellaso in 1553. (It was later misattributed to Blaise de Vigenère, who invented a different but similar cipher around the same time.) It uses a word as its key, rather than a single letter. Each letter of the plaintext is shifted, just as in a Caesar cipher, but the same shift is not used every time. Instead, each letter of the key specifies a shift, and they are used in order and cycled. This can be visualized by writing the key, repeating, underneath the plaintext. For example, let's imagine again that we're encrypted HAPPYBIRTHDAY, this time with the key BAD. (In practice, a longer word would probably be used.) This results in the following computation:

plaintext:	H	A	P	P	Y	B	I	R	T	H	D	A	Y
key:	B	A	D	B	A	D	B	A	D	B	A	D	B
ciphertext:	I	A	S	Q	Y	E	J	R	W	I	D	D	Z

In our previous example of a Caesar cipher, we used the key of D. That would send E to H, making H likely to be the most common letter in a long ciphertext. But here an E is sometime sent to H, but also sometimes to E and sometimes to F. This balances out the frequency of each letter in the resulting message, making straightforward frequency analysis not work. (It's still breakable, but before we get to that we need to get code to actually encrypt and decrypt using a Vigenère cipher.)

`vigenere_enc`

Write a function that computers the encryption algorithm for the Vigenère cipher. It should take two inputs, the first being a (simplified) plaintext, and the second being they key, represented as a string of capital letters. The output should be the correct ciphertext encrypting that plaintext under that key.

`vigenere_enc("HAPPYBIRTHDAY", "BAD")` returns `"IASQYEJRWIDDZ"`.

`vigenere_dec`

This is the decryption function that reverses the encryption above. It takes two inputs, a ciphertext and a key, and it returns the original plaintext.

`vigenere_dec("IASQYEJRWIDDZ", "BAD")` returns `"HAPPYBIRTHDAY"`.

Part 4: Breaking the Vigenère cipher

While cryptanalysts occasionally deciphered particular methods, no general attack on the Vigenère cipher was published for hundreds of years. The first published attack was by Kasiski in 1863, though Charles Babbage is known to have also developed an attack as early as 1854. In the American Civil War, the Confederacy regularly used Vigenère ciphers to send messages, but the Union regularly broke their encryption. This didn't stop Scientific American from declaring the cipher "impossible of translation" in 1917, and in general the Vigenère cipher has a reputation that greatly exaggerates its security.

`split_string`

This is a simple utility function. The Vigenère cipher essentially applies several different Caesar ciphers to a plaintext, but applies each one to only one subset of the letters. `split_string` divides the plaintext into those relevant parts. The function takes two inputs, a string and an integer. The integer can be thought of as the length of the cipher's key, and it specifies the number of parts that the string should be divided into. The output should be a list of that many strings, each equal to the letters that would be encrypted using a particular letter of the key.

`split_string("HAPPYBIRTHDAY", 3)` returns `["HPIHY", "AYRD", "PBTA"]`.

Note that the first string in the output contains all the letters that would be encrypted using the first letter of the key (assuming the key is 3 letters long). The second element of the list is a string of the letters that would be encrypted under the second letter of the key, and so forth.

vigenere_break_for_length

This function takes three inputs, the ciphertext, the length of the key, and a dictionary of frequencies in standard English. It should break the cipher and return a list of two elements, the key and the plaintext. It is not a general break because it assumes we have somehow learned the length of the key, but it is a step in the right direction.

To break the ciphertext, you should simply use the same frequency analysis we used to break a Caesar cipher. After all, the parts that `split_string` will return are each a set of letters from standard English encrypted under a Caesar cipher. Carry out that attack separately on each part and then combine the results.

`vigenere_break_for_length("KDSSBELUWKGDB", 3, english_freqs)` would return `["BAD", "HAPPYBIRTHDAY"]`.
(This test won't work because the message is too short for frequency analysis to be reliable, but it should give you an indication of the specifications.)

vigenere_break

This function takes three inputs, the ciphertext, a maximum length of the key, and a dictionary of frequencies in standard English. It should break the encryption and return the key and plaintext, as long as the real key is of length less than or equal to the maximum length given. To do this, simply use the attack above assuming each possible length of key. Then for each resulting plaintext, compare the letter frequencies to those of standard English. Whichever is closest you can assume is the correct key length. If multiple key lengths result in the same distance to standard English frequencies, return the shorter key.

`vigenere_break("KDSSBELUWKGDB", 10, english_freqs)` would return `["BAD", "HAPPYBIRTHDAY"]`.
(This test won't work because the message is too short for frequency analysis to be reliable, but it should give you an indication of the specifications.)

Again, you can use the text in `hitchhiker's.txt` to test your encryption and decryption functions, and you can break the encrypted version to test your breaking functions. We have also included `mystery2.txt`, which contains a ciphertext encrypted under a Vigenère cipher. You should now be able to break it.

Part 5: A working substitution cipher

We're going to take a step backward historically and move to *substitution ciphers,* a generalization of Caesar ciphers. (We're going out of order because substitution ciphers are going to require some different techniques.) A substitution cipher, like a Caesar cipher, replaces each letter in the plaintext with a particular letter in the ciphertext. However, the choices of which letter is replaced by what are independent, rather than all letters being shifted by a certain amount. A key specifies a particular replacement for each letter. For example, say we have the following key (written under the alphabet for convenience):

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

According to this key, every instance of A is replaced by a J, every instance of B replaced by a Z, and so forth. So if we again encrypt HAPPYBIRTHDAY, we get the following result:

```
plaintext:   H A P P Y B I R T H D A Y
ciphertext:  L J D D O Z P C Y L S J O
```

This means there are a lot more possible keys (26!, or about 2^{88}), and it is infeasible to try them all.

sub_gen_key

This is a function that generates a random key for your cipher, something like the key above. It takes no inputs and its output should be a string of the 26 capital letters in the alphabet, arranged in a random order. All orders should be equally likely.

sub_enc

This is the encryption function. It takes two inputs, a simplified string and a key, and it outputs a valid ciphertext. They key should be interpreted as above, meaning that the first letter is the ciphertext's replacement for A, the second is the replacement for B, and so forth.

`sub_enc("HAPPYBIRTHDAY", "JZMSKIRLPWVNUEXDTCGYFQBHOA")` returns `"LJDDOZPCYLSJO"`.

vigenere_dec

This is the decryption function that reverses the encryption above. It takes two inputs, a ciphertext and a key, and it returns the original plaintext.

`sub_dec("LJDDOZPCYLSJO", "JZMSKIRLPWVNUEXDTCGYFQBHOA")` returns `"HAPPYBIRTHDAY"`.

Part 6: Breaking the substitution cipher

Finally, we want to break the substitution cipher. This isn't that hard – people do these by hand all the time as "cryptogram" puzzles in newspapers. The problem is that doing it well requires some knowledge of English. If all you know is letter frequencies, you can get a rough guess. The most common letter in a large piece of text is probably E. The next most common are probably A, I, O, and T. But A, I, O, and T have very similar frequencies, so you can't rely on frequency alone to distinguish them (unless your text is extremely long). Humans look at things like which letters appear twice in a row – lots of words have OO or TT in them, but very few have AA or II. They also recognize particular words or other common patterns like ING or ED endings. We're going to need our program to do something like this.

`count_trigrams`

It turns out that we don't need to consider full words, but just small patterns that appear. We'll look at trigrams, series of three letters. Some trigrams like THE and ING are very common in English, whereas others like QAZ are very rare. We will use these frequencies to determine what keys are reasonable candidates.

This function should take as input a string and output a dictionary of trigrams. Each key should be a trigram, something like `"THE"`, and the corresponding value should be the number of times it appears in the string. It is ok if trigrams that don't appear in the string at all also don't appear in the resulting dictionary.

Once this is done, you can uncomment the lines of code that compute english_trigrams. This uses the above counts and then normalizes the dictionary. (This is why `normalize` had to work for dictionaries other than just those with the 26 letters as keys.)

`map_log`

Given a string, we want to compute a score for that string equal to the "probability" that the trigrams in that string would be the trigrams in a random string. (This isn't quite right mathematically because it ignores that trigrams overlap, but it'll be good enough.) If the trigrams in a string are $t_1, t_2, \ldots, t_n$, then the score would be

$$score = t_1 \times t_2 \times t_3 \times \ldots \times t_n.$$

The problem is that this is the product of lots of very low numbers, so we will have lots of floating point errors, and those errors will be big enough to matter. So instead, we take the logarithm (base e) of both sides. This changes the multiplications of tiny numbers to additions of reasonable numbers.

$$\ln(score) = \ln(t_1) + \ln(t_2) + \cdots + \ln(t_n)$$

We can then compare the log-scores instead of the original scores. (They aren't the same, but the comparison between the scores of two strings will still result in the same decision on which string has the better score.)

In order to do this computation, we're going to just always use the logs of frequencies instead of the frequencies, so we need to write `map_log`, which takes as input a dictionary of frequencies and replaces every value with the natural log of that value.

You can then uncomment the line that applies this function to our dictionary of trigram frequencies.

If everything is working correctly, you'll find that `english_triagrams["THE"]` has a value of roughly -3.8913, while `english_triagrams["ARH"]` has a value of roughly -9.9034. This shows that the trigram THE is much more common in English than the trigram ARH.

`trigram_score`

Given a string, we want to compute the above (log-)score for that string. This is simply the sum of the log-frequency (in standard English) of each trigram in the string. (See the formula above.) Low scores indicate that a string has more unusual trigrams in it. This function should take two inputs, a string and a dictionary of trigram log-frequencies, and it should return the score of that string. If a trigram isn't present in the dictionary (meaning that it never appeared in our large sample of English text), then count its log-frequency as -15. (A frequency of 0 would result in a log-frequency of negative infinity, but it's not *impossible* that the decryption contains a trigram not in our English text. It's just very unlikely.)

You should find that `trigram_score("HAPPYBIRTHDAY", english_trigrams)` returns roughly -100.829.

`sub_break`

This is the function that finally breaks a substitution cipher, taking as input a ciphertext and a dictionary of trigram log-frequencies. It works a little differently from what we did before. The basic algorithm is as follows:

- Pick a random key as a starting guess. Decrypt the ciphertext with that key and compute the trigram score of that decryption.

- Pick two random letters in the key and swap them. (All choices of two letters should be equally likely.) This gives a new candidate key.
- Decrypt the ciphertext with the candidate key. If that decryption has a higher score than the decryption under the current key guess, replace the current key guess with the candidate key.
- Keep trying new swaps. These will often not be improvements, but sometimes they will be and your guessed key will get closer to the real key.
- If at some point you try 1000 random swaps without finding one that improves the score of the decryption, then assume you are done. Return your current key guess and the corresponding decryption.

You should implement the algorithm above. As always, you can use an encryption of `hitchhiker's.txt` to check that your encryption, decryption, and breaking functions work. When you are done you should be able to decrypt `mystery3.txt`, which has been encrypted using a substitution cipher.