Introduction

Bookmark this page

Encryption is the process of obscuring information to make it unreadable without special knowledge. For centuries, people have devised schemes to encrypt messages - some better than others - but the advent of the computer and the Internet revolutionized the field. These days, it's hard not to encounter some sort of encryption, whether you are buying something online or logging into a shared computer system. Encryption lets you share information with other trusted people, without fear of disclosure.

A cipher is an algorithm for performing encryption (and the reverse, decryption). The original information is called plaintext. After it is encrypted, it is called ciphertext. The ciphertext message contains all the information of the plaintext message, but it is not in a format readable by a human or computer without the proper mechanism to decrypt it; it should resemble random gibberish to those for whom it is not intended.

A cipher usually depends on a piece of auxiliary information, called a key. The key is incorporated into the encryption process; the same plaintext encrypted with two different keys should have two different ciphertexts. Without the key, it should be difficult to decrypt the resulting ciphertext into readable plaintext.

This assignment will deal with a well-known (though not very secure) encryption method called the Caesar cipher. Some vocabulary to get you started on this problem:

- *Encryption* - the process of obscuring or encoding messages to make them unreadable until they are decrypted
- *Decryption* - making encrypted messages readable again by decoding them
- *Cipher* - algorithm for performing encryption and decryption
- *Plaintext* - the original message
- *Ciphertext* - the encrypted message. Note: a ciphertext still contains all of the original message information, even if it looks like gibberish.

The Caesar Cipher

The idea of the Caesar Cipher is to pick an integer and shift every letter of your message by that integer. In other words, suppose the shift is k . Then, all instances of the i-th letter of the alphabet that appear in the plaintext should become the (i+k)-th letter of the alphabet in the ciphertext. You will need to be careful with the case in which i + k > 26 (the length of the alphabet). Here is what the whole alphabet looks like shifted three spots to the right:

```
Original:  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
 3-shift:  d e f g h i j k l m n o p q r s t u v w x y z a b c
```

Using the above key, we can quickly translate the message "happy" to "kdssb" (note how the 3-shifted alphabet wraps around at the end, so x -> a, y -> b, and z -> c).

Note!! We are using the English alphabet for this problem - that is, the following letters in the following order:

```
>>> import string
>>> print string.ascii_lowercase
Abcdefghijklmnopqrstuvwxyz
```

We will treat uppercase and lowercase letters individually, so that uppercase letters are always mapped to an uppercase letter, and lowercase letters are always mapped to a lowercase letter. If an uppercase letter maps to "A", then the same lowercase letter should map to "a". Punctuation and spaces should be retained and not changed. For example, a plaintext message with a comma should have a corresponding ciphertext with a comma in the same position.

plaintext	shift	ciphertext
'abcdef'	2	'cdefgh'
'Hello, World!'	5	'Mjqqt, Btwqi!'
''	any value	''

We implemented for you two helper functions: `load_words` and `is_word`. You may use these in your solution and you do not need to understand them completely, but should read the associated comments. You should read and understand the helper code in the rest of the file and use it to guide your solutions.

Getting Started

To get started, download the ps6.zip file. Extract it to your working directory. The files inside are:

- ps6.py - a file containing three classes that you will have to implement.
- words.txt - a file containing valid English words (should be in the same folder as your ps6..py file).
- story.txt - a file containing an encrypted message that you will have to decode (should be in the same folder as your ps6..py file).

This will be your first experience coding with classes! We will have a `Message` class with two subclasses `PlaintextMessage` and `CiphertextMessage` .

Problem 1 - Build the Shift Dictionary and Apply Shift

20.0/20.0 points (graded)

The `Message` class contains methods that could be used to apply a cipher to a string, either to encrypt or to decrypt a message (since for Caesar codes this is the same action).
In the next two questions, you will fill in the methods of the `Message` class found in `ps6.py` according to the specifications in the docstrings. The methods in the `Message` class already filled in are:

- `__init__(self, text)`
- The getter method `get_message_text(self)`
- The getter method `get_valid_words(self)`, notice that this one returns a copy of `self.valid_words` to prevent someone from mutating the original list.

In this problem, you will fill in two methods:

1. Fill in the `build_shift_dict(self, shift)` method of the `Message` class. Be sure that your dictionary includes both lower and upper case letters, but that the shifted character for a lower case letter and its uppercase version are lower and upper case instances of the same letter. What this means is that if the original letter is "a" and its shifted value is "c", the letter "A" should shift to the letter "C".
   If you are unfamiliar with the ordering or characters of the English alphabet, we will be following the letter ordering displayed by`string.ascii_lowercase` and `string.ascii_uppercase`:
   ```
   >>> import string
   >>> print(string.ascii_lowercase)
   abcdefghijklmnopqrstuvwxyz
   >>> print(string.ascii_uppercase)
   ABCDEFGHIJKLMNOPQRSTUVWXYZ
   ```
   A reminder from the introduction page - characters such as the space character, commas, periods, exclamation points, etc will *not* be encrypted by this cipher - basically, all the characters within `string.punctuation`, plus the space (`' '`) and all numerical characters (0 - 9) found in `string.digits`.

2. Fill in the `apply_shift(self, shift)` method of the `Message` class. You may find it easier to use `build_shift_dict(self, shift)`. Remember that spaces and punctuation should not be changed by the cipher.

Paste your implementation of the `Message` class in the box below.

Problem 2 - PlaintextMessage

15.0/15.0 points (graded)

For this problem, the graders will use our implementation of the `Message` class, so don't worry if you did not get the previous parts correct.
`PlaintextMessage` is a subclass of `Message` and has methods to encode a string using a specified shift value. Our class will always create an encoded version of the message, and will have methods for changing the encoding.
Implement the methods in the class `PlaintextMessage` according to the specifications in ps6.py. The methods you should fill in are:

- `__init__(self, text, shift)`: Use the parent class constructor to make your code more concise.
- The getter method `get_shift(self)`
- The getter method `get_encrypting_dict(self)`: This should return a COPY of self.encrypting_dict to prevent someone from mutating the original dictionary.
- The getter method `get_message_text_encrypted(self)`
- `change_shift(self, shift)`: Think about what other methods you can use to make this easier. It shouldn’t take more than a couple lines of code.

Paste your implementation of the entire `PlaintextMessage` class in the box below.

Problem 3 - CiphertextMessage

15.0/15.0 points (graded)

For this problem, the graders will use our implementation of the `Message` and `PlaintextMessage` classes, so don't worry if you did not get the previous parts correct.
Given an encrypted message, if you know the shift used to encode the message, decoding it is trivial. If `message` is the encrypted message, and `s` is the shift used to encrypt the message, then `apply_shift(message, 26-s)` gives you the original plaintext message. Do you see why?

The problem, of course, is that you don’t know the shift. But our encryption method only has 26 distinct possible values for the shift! We know English is the main language of these emails, so if we can write a program that tries each shift and maximizes the number of English words in the decoded message, we can decrypt their cipher! A simple indication of whether or not the correct shift has been found is if most of the words obtained after a shift are valid words. Note that this only means that most of the words obtained are actual words. It is possible to have a message that can be decoded by two separate shifts into different sets of words. While there are various strategies for deciding between ambiguous decryptions, for this problem we are only looking for a simple solution.

Fill in the methods in the class `CiphertextMessage` acording to the specifications in ps6.py. The methods you should fill in are:

- `__init__(self, text)`: Use the parent class constructor to make your code more concise.
- `decrypt_message(self)`: You may find the helper function `is_word(wordlist, word)` and the string method `split()` useful. Note that `is_word` will ignore punctuation and other special characters when considering whether a word is valid.

Problem 3 - CiphertextMessage

15.0/15.0 points (graded)

For this problem, the graders will use our implementation of the `Message` and `PlaintextMessage` classes, so don't worry if you did not get the previous parts correct.
Given an encrypted message, if you know the shift used to encode the message, decoding it is trivial. If `message` is the encrypted message, and `s` is the shift used to encrypt the message, then `apply_shift(message, 26-s)` gives you the original plaintext message. Do you see why?

The problem, of course, is that you don’t know the shift. But our encryption method only has 26 distinct possible values for the shift! We know English is the main language of these emails, so if we can write a program that tries each shift and maximizes the number of English words in the decoded message, we can decrypt their cipher! A simple indication of whether or not the correct shift has been found is if most of the words obtained after a shift are valid words. Note that this only means that most of the words obtained are actual words. It is possible to have a message that can be decoded by two separate shifts into different sets of words. While there are various strategies for deciding between ambiguous decryptions, for this problem we are only looking for a simple solution.

Fill in the methods in the class `CiphertextMessage` acording to the specifications in ps6.py. The methods you should fill in are:

- `__init__(self, text)`: Use the parent class constructor to make your code more concise.
- `decrypt_message(self)`: You may find the helper function `is_word(wordlist, word)` and the string method `split()` useful. Note that `is_word` will ignore punctuation and other special characters when considering whether a word is valid.