420-LCU-05 Programming in Python - Assignment 4

November 16, 2017

Here is a reminder of the general requirements for these assignments:

1. **Identification section** Do this for *every* Python file in every assignment in this course. This section must be either in a comment, with a '#' preceding each line, or enclosed within triple quotes ("'). The grader and I need this section for the *accurate processing of your assignment*. Assignments missing this may lose up to 5% of the total mark.

   Please note that you should also include your ID number after your name!

   Example:

   ```
   """
   Justin Trudeau, 1122334
   420-LCU Computer Programming, Section 2
   Sunday, February 31
   R. Vincent, instructor
   Assignment 1
   """
   ```

2. Your submission for this assignment will be one Python file, there is no need to submit a ZIP file.

3. Be sure to respect other instructions specified in the assignment. Part of each assignment is to correctly follow the instructions as closely as possible.

4. Remember that you *must* include comments in your code. You do not have to have a comment on every line, but you should have roughly one or two comments per method or function.

5. Any method, class, or function you create must have a docstring.

Introduction[1]

The process of encryption hides a message in order to make it difficult to read for anyone who does not know some secret or password. Today there are many excellent encryption methods available, but some form of encryption has existed for centuries. To start us off, here are a few relevant vocabulary words:

- *Encryption* - Encoding a message to make it unreadable.
- *Decryption* - Decoding a message to make it readable again.
- *Cipher* - An algorithm for encryption and decryption.
- *Plaintext* - The original message.
- *Ciphertext* - An encrypted message. Remember that even though the encrypted message may be scrambled, all of the information from the original message is still retained.

We have provided three files in this assignment:

[1] This assignment is adapted from one used in the MITx 6.00.1x course.

- `a4_cipher.py` - the source code you will modify.
- `story.txt` - a message your program will decode.
- `words.txt` - a list of English words.

You should make sure all three of these files are in the same folder before you test your work.

The Caesar Cipher

A *Caesar Cipher* is a very simple, and very easy to crack, way of encrypting a message. The concept is to pick an integer j between 1 and 25, and exchange every letter of a message with the letter j positions later in the alphabet. Each letter i of the alphabet is transformed into letter $i + j$. If $i + j$ is greater than 26, we have to wrap back to the beginning of the alphabet. For example, for a shift of 3 positions we get this table:

Normal:	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 this table, we can encrypt the message "hello" as "khoor", and "zebra" as "cheud".

As you can see, we are using the English alphabet here. You can get the uppercase and lowercase alphabet by importing them from the `string` module:

```
>>> import string
>>> print(string.ascii_lowercase)
abcdefghijklmnopqrstuvwxyz
>>> print(string.ascii_uppercase)
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

For this assignment you must take care that your encryption and decryption code retains the case of the original message. You should not translate spaces, digits or punctuation.

Here is a table that shows some examples:

Plaintext	Shift	Ciphertext
Hello, World!	4	Lipps, Asvph!
Good morning	5	Ltti rtwsnsl
Nov. 1st	1	Opw. 1tu

We have provided two global functions, `load_words` and `get_story_string()`, some testing code, and three incomplete class definitions, that you will use in your solution. You do not have to fully understand these functions, but you should look at the docstrings and understand how to use them.

Your job is to complete the three classes that represent our cipher algorithm. The first, `Message`, will represent all messages. From that class we will derive two specialized classes, `PlaintextMessage` and `CiphertextMessage`.

Exercise 1 - The Message class

The `Message` class contains methods that are used to both decrypt or encrypt, since the Caesar cipher uses essentially the same operation for both. The `Message` class already contains the following methods:

- `__init__(self, text)`
- `get_message_text(self)`

You must fill in two methods:

1. Fill in the `build_shift_dict(self, shift)` method. This method builds a dictionary to be used in the cipher. The dictionary you create must contain keys for both lower case and upper case letters, so that lower case letters map onto other lower case letters and upper case letters map onto upper case characters. Use the `string.ascii_lowercase` and `string.ascii_lowercase` mentioned above.
2. Fill in the `apply_shift(self, shift)` method, which applies the cipher to the message text. This method will use the `build_shift_dict(self, shift)` method.

When you fill in these methods, you may need to remove or change the single `return` statement we used as a placeholder in the methods.

Exercise 2 - The PlaintextMessage class

`PlaintextMessage` is derived from the `Message` class and has methods to encrypt the message text using a specific shift.

We've already provided three methods. Read their docstrings for details:

1. `get_shift(self)`
2. `set_shift(self, shift)`
3. `get_message_text_encrypted(self)`

For this class, you just need to complete the `__init__(self, text, shift)__` method. It should be very simple - use the base class implementation of `__init__(self, text)` and the `set_shift(self, shift)` method.

Exercise 3 - The CiphertextMessage class

For this part, you need to create the CiphertextMessage class. In the Caesar cipher, if you know the shift used to encrypt the message, it is easy to decrypt the message. If you don't know the shift, the problem is harder. The good news, from the point of view of someone trying to crack the cipher, is that there are only 26 possible shift values, and it is easy to try them all!

To decide which of the shift values is the best, you need a way to determine when you have correctly decoded the string. You can do this by keeping track of the number of actual English words present in the decrypted text. The shift value that yields the most English words is probably the correct shift.

We have provided one important method:

- `is_word(self, word)` - this method will check whether the string `word` is actually a word in English. It does so by checking to see whether the word is present in the `valid_words` attribute.

You must complete two methods:

1. `__init__(self, text)`, which will use the base class implementation, and also use the `load_words` function mentioned above.

2. `decrypt_message(self)`, which tries all 26 possible shift values, and determines which one is the "best" shift value for decrypting the message. Hint: You will probably use `str.split()`.

Testing

We have provided some test code that will test a couple of your methods, and then test the behaviors of your classes. If your code is correct, this program should complete and print out the decoded contents of `story.txt`.

Once you have tested your code and verified your identification section, you can then submit your completed `a4_caesar.py` to Omnivox.