420-LCU-05 Programming in Python - Assignment 4

April 12, 2024

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:

   ```
   """
   Radia Perlman, 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 some comments in each function. Obviously, the longer the function, the more comments I expect to see.

5. Any function you create must have a docstring. However, I have already provided docstrings for the assigned functions, so you will only have to add them if you define additional functions.

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 algorithms 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.

I have provided four files in this assignment:

- `caesar.py` - the source code you will modify.

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

- `a4test.py` - test code that verifies your code is correct.
- `message.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, at least for the English language, 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 a shift of 3 positions would give Table 1 (assuming lower-case letters only). Using Table 1, one can encrypt the message “hello” as “khoor”, and

Table 1: Caesar cipher table for shift of 3

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

“zebra” as “cheud” by reading the original letters from the top row and substituting the corresponding letter from the bottom row.

As a convenience, you can get the uppercase and lowercase English 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. Table 2 shows some examples illustrating this.

Table 2: Caesar cipher examples

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

In the provided file `caesar.py` you will find the completed function `read_word_list()` as well as several incomplete functions. You must not modify the provided functions, except for those you are specifically required to complete.

I have also provided header lines and docstrings for the five functions you are required to create. Be sure to read the docstrings for additional details about the requirements for the assignment!!

Finally there are two named values, `N_LETTERS` and `VALID_WORDS` that you will need to use in your functions. The first, `N_LETTERS`, is just a handy symbol that contains the `int` giving the number of letters in the English alphabet. The name `VALID_WORDS` is a Python `frozenset` that contains all of the words from the `words.txt` file. A `frozenset` is an immutable set of keys, sort of like an immutable dictionary without the values.

Exercise 1: `create_shift_table(shift)`

This function builds a dictionary (a Python `dict`) to be used in the cipher. This dictionary should mimic the table shown above: each key will be a letter of the alphabet, and each value will be the letter of the alphabet after the shift is applied. The dictionary 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 letters. Use the `string.ascii_lowercase` and `string.ascii_uppercase` mentioned above.

The `shift` must be between 0 and `N_LETTERS-1`, inclusive. Otherwise your function should return `None`. Be sure to use `N_LETTERS` where appropriate.

Exercise 2: `apply_shift(text, shift)`

This function applies the cipher to the `text`, by creating a shift table for the Caesar cipher of `shift` places, then iterating over the characters in `text` to compute the encrypted (or decrypted) text. The function will return the resulting encrypted string. This function, which may be only a few lines in length, will use your `create_shift_table(shift)` function.

Note that, because of the simplicity of the Caesar cipher, the same function can be used for both encryption and decryption. Consider the following code:

```
# Given:
#   plaintext is some string to be encrypted
#   shift is a number between 0 and N_LETTERS-1
ciphertext = apply_shift(plaintext, shift)
resulttext = apply_shift(ciphertext, N_LETTERS - shift)
# resulttext should equal plaintext
```

When the last line has executed, `resulttext` should be identical to the original `plaintext`. Note that this also implies that for a `shift` of 13, the exact same process will both encrypt or decrypt.

Exercise 3: `encrypt_text(original_text, shift)`

This function uses the plain text message `original_text` and the `shift` to calculate the encrypted text for the given message. It should return the encrypted message.

In practice, this function will be very simple. It will call your `apply_shift()` function. This is quite common in real projects. The `apply_shift()` function is used *internally* by the Caesar cipher module, but it is not intended to be used elsewhere. The `encrypt_text()` function is the *public* interface to the same functionality.

Exercise 4: `is_word(word)`

This function will check whether the string `word` is actually a word in English. It does so by checking whether the word is present in the `VALID_WORDS` set.

The `word` as given may include punctuation or spaces at the beginning or end. It may also use either upper or lower case letters. You will want to use the `strip()` method to remove possible punctuation or spaces. You can get strings containing all punctuation characters and space characters from `string.punctuation` and `string.whitespace`.

This function should be only a couple of lines of code. Remember that a `set` or `frozenset` works kind of like a dictionary. You can check if a value is contained in `VALID_WORDS` using the `in` operator.

Exercise 5: `decrypt_message(text)`

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, but still no trouble for a computer. The good news, from the point of view of someone trying to crack the cipher, is that there are only 26 possible shift values (`0...N_LETTERS-1`), 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 correct English words present in the decrypted text. The shift value that yields the most English words is probably the correct shift.

You must complete the function `decrypt_message(text)`, which will try all `N_LETTERS` possible shift values, and determine which one is the "best" shift value for decrypting the message.

You will need to use your function `is_word(word)` to check for English words. Note that `is_word(word)` only works with *individual* words. If you call `is_word("The sun is up")` it will return `False`, for example. Hint: You will probably use `str.split()`.

You will also need to use your `apply_shift()` function to try each shift value.

Your function must return a `tuple` containing two values: the integer giving the best decryption shift, and a string containing the actual decrypted text.

Be sure to use `N_LETTERS` where appropriate.

Testing

I have provided some code, `a4test.py`, that will test your functions. Because `caesar.py` is not complete as provided, the tests will fail until you provide correct implementations. Once you have finished your functions, this program should run without error and print out the decoded contents of `message.txt`.

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