Lab overview video. The following videos provide an overview to this lab assignment:
https://williams.hosted.panopto.com/Panopto/Pages/Viewer.aspx?pid=55f5e97f-8fda-4dcd-ac11-aba0012c79d9

This week, we'll look at simple Caesar ciphers for encrypting and decrypting messages. Encryption is the process of obscuring or encoding messages to make them unreadable, decryption makes encrypted messages readable again by decoding them. A cipher is an algorithm for performing encryption and decryption. We'll call an un-encrypted, readable message plaintext and a (probably unreadable) encrypted message ciphertext.

The idea of the Caesar Cipher is to pick an integer and shift every letter of the plaintext forward in the alphabet 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. We need to be careful, of course, to handle the case when a letter is shifted beyond the end of the alphabet. In that case, we wrap around and continue shifting from the beginning of the alphabet. So, for example, if the plaintext is 'jazz', and we shift by 1 letter, the result would be 'kbaa'. Because of this wrap-around, it’s common to call the shift a rotation.

We will treat uppercase letters and lowercase letters as two different cases of the same problem. Uppercase letters are always encrypted as uppercase letters, and lowercase letters are always encrypted as lowercase letters. For the purposes of this lab, punctuation and spaces should be retained and not changed. For example, the plaintext 'Hello world!', shifted 2 becomes the ciphertext 'Jgnnq yqtnf!'.

Here are some examples:

<table>
<thead>
<tr>
<th>plaintext</th>
<th>shift</th>
<th>ciphertext</th>
</tr>
</thead>
<tbody>
<tr>
<td>'abcdef'</td>
<td>2</td>
<td>'cdefgh'</td>
</tr>
<tr>
<td>'Hello, world!'</td>
<td>4</td>
<td>'Lipps, asvph!'</td>
</tr>
<tr>
<td>'OCT: two keg.'</td>
<td>24</td>
<td>'MAR: rum ice.'</td>
</tr>
<tr>
<td>' '</td>
<td>any</td>
<td>''</td>
</tr>
</tbody>
</table>

Getting Started. First, remember to connect to the Williams network with the VPN software. Now, clone the starter repository for this week’s lab, using the process you followed for our previous remote labs. Go to https://evolene.cs.williams.edu, navigate to this week’s repository, click Clone and select the Copy URL to clipboard button under the Clone with HTTPS text. Return to the Terminal/Command Prompt, navigate to your cs134 directory, and type git clone and paste the URL you just copied, followed by the name of the lab, e.g.:

git clone https://evolene.cs.williams.edu/cs134-s20/lab08/22xyz3.git lab08/

where your CS username replaces 22xyz3.

You will find the file, cipher.py, along with a few text files in the lab folder. words.txt contains a dictionary of common English words. We’ve also included story.txt, a secret story you may find you wish to decrypt.
This week’s tasks.

In the file `cipher.py` you will find a few standalone “helper” methods and the beginnings of three class definitions. The classes are:

- **Message.** Message objects hold text strings and have functions that support shifting text through the alphabet and counting words.

- **Plaintext.** The Plaintext objects represent messages to be encrypted. These objects keep track of a shift count and can encrypt the plaintext. The class extends the Message class, so you inherit the ability to rotate and count words in the plaintext.

- **Ciphertext.** The Ciphertext objects are Messages that are meant to be decrypted. They support manual and automatic decryption into Plaintext objects.

You may find the following helper methods useful (all but one are complete):

- **canonical(word).** Takes a string, word, removes non-letters and converts the result into lower case. It’s useful for isolating words in typical text.

- **loadWords(filename).** Reads a word list from a file. Our code reads our words.txt file, providing you with a list of English words, called dictionary.

- **isWord(word, wordList=dictionary).** Returns True if and only if a word is found in a wordList. Words are converted into their canonical forms before the word list is checked.

- **loadStory(file).** Reads in a story from a file and returns it as a (long) string. By default, it reads our secret (and thrilling) story from "story.txt".

- **rotateLetter(c,n).** This function takes and encrypts a single character, based on an integer shift amount, n. This is the only helper function that needs to be completed.

**Task 1.** Finish the `rotateLetter(c,n)` helper function. As mentioned above, this function computes and returns the encryption of a single character. Letters are mapped to other letters that appear n positions later in the alphabet, wrapping around at the end. Uppercase and lowercase letters should be mapped amongst themselves. Non-letters should be returned without change.

```
>>> rotateLetter('a', 2)
'c'
>>> rotateLetter('H', 4)
'L'
>>> rotateLetter('!', 5)
'!
```

Lecture videos provide guidance with this mapping process.

**Task 2. Implementing the Message class.** Add appropriate doctests, and complete these method definitions in the Message class:

(a) **__init__(self, text).** This method initializes an object of the class Message. You should add _text to the _slots_ list and initialize it to the text passed in here.
(b) `text(self)`. This is a property method that returns the object’s `.text` attribute. Notice the @property annotation preceding the method definition. Here is an example of how this method is invoked:

```python
>>> msg = Message('goodbye')
>>> msg.text
'goodbye'
```

(c) `rotate(self, shift)`. This method applies the Caesar Cipher to the message’s text. It returns a string that is `.text`, shifted forward and around the alphabet by some number of positions determined by the integer, `shift`. Here’s how it works.

```python
>>> Message('hello').rotate(2)
'jgnnq'
>>> Message('Cheer').rotate(7)
'Jolly'
```

(d) `wordCount(self)`. This is a property method that computes and returns the number of words that appear in the message’s text. (You will find the `isWord` helper method helpful here.)

```python
>>> msg = Message("Hello, world!")
>>> msg.wordCount
2
>>> Message("Who hates covid-19?").wordCount
2
```

Remark. This point is a good time to take a break. Check to make sure the `Message` class works as expected. Try typing the following commands into python3 and see what happens:

```python
>>> from cipher import Message
>>> msg = Message("Hello, world!")
>>> msg.text
'Hello, world!
>>> msg.wordCount
2
>>> msg.rotate(4)
'Lipps, wsvph!'
```

Only move onto the next task when the `Message` class has been successfully completed.

Task 3. Implementing the `Plaintext` class. `Plaintext` is a subclass of `Message` and inherits its attributes and methods. In addition to those, add appropriate doctests, and complete these method definitions in the `Plaintext` class:

(a) `__init__(self, text, shift)`. This method is invoked when we create an object of the class `Plaintext` and initializes the object’s inherited attribute `.text` and its additional attribute `.shift` with the arguments `text` and `shift`. 
Because every Plaintext is a Message, we initialize its Message-related attributes by calling the __init__ method from the superclass, Message. We have given you that one piece of code:

```python
super().__init__(text)
```

All that is left is to declare the _shift attribute and give it an initial value.

(b) shift(self). This is a property method which is used to safely access self._shift outside of the class.

```python
>>> msg = Plaintext('goodbye', 5)
>>> msg.shift
5
```

(c) shift(self, shiftValue). This is the shift property's setter method. It is used to safely modify the _self attribute from outside the class. Notice the @shift.setter annotation preceding the method definition. Make sure the change in the shift value is reflected in any Plaintext attribute that depends on shift.

```python
>>> msg = Plaintext('goodbye', 5)
>>> msg.shift = 3
>>> msg.shift
3
```

(d) encrypted(self). This is a property method that returns the result of shifting the Plaintext's text by its current shift amount. You have a choice here. (1) If you wish, you can compute and store the encrypted text as an attribute of the class. (2) Alternatively, you can compute it "on the fly", whenever this method is called. Because the attributes are not directly accessible by the user of this class, that decision is hidden from view. Just be careful that you always get the correctly encrypted value, even after the shift property is changed:

```python
>>> msg = Plaintext('hello', 1)
>>> msg.shift = 2
>>> msg.encrypted
'jgnnq'
```

Remark. Break time! Check to make sure the Plainclass class works as expected. You should be able to perform all the little experiments using your new code, as described above. Your mother would be very proud (ask her). But something subtle is happening here: because Plaintext objects inherit Message attributes and functionality, we should be able use Plaintext methods in the same way we used Message objects:

```python
>>> from cipher import Plaintext
>>> zoom = Plaintext("Hello Iris & Shikha!",0)
>>> zoom.text
'Hello Iris & Shikha!'
>>> zoom.wordCount
2 # why?
>>> zoom.rotate(4)
```
Only move onto the next task when you're happy with Plaintext.

Task 4. Implementing the Ciphertext class. Ciphertext is a subclass of Message. It inherits all of Message's attributes and methods, including rotate and wordCount. As you complete this task, think about if you need any additional attributes. This task requires you to complete the method definitions of the Ciphertext class:

(a) __init__(self, text). This method is invoked when we create an object of the class Ciphertext and initializes the object's attribute text with the argument text. Follow the example of Plaintext and make use of Message's initializer method. If you declare any additional attributes, it will be important to initialize them here.

(b) decrypt(self, unshift). This method is used to test possible decryptions of the underlying text. Because we're decrypting, the process is a rotation backwards through the alphabet. Fortunately, a backwards rotation is the same as a forward rotation by shift = 26-unshift positions. Compute the appropriate rotation and construct and return a Plaintext object initialized with the result of the rotation and an appropriate shift. Consider checking this by using the interactive python:

```python
>>> from cipher import Ciphertext
>>> cipher = Ciphertext('jgnnq')
>>> plain = cipher.decrypt(2)
>>> plain.text
'hello'
>>> plain.shift
2
>>> plain.encrypted
'jgnnq'
```

Remark. This point is a good time to take a break and test your work. Do all your methods so far pass the doctests? Stage, commit, and push your work! If you've gotten to this point in the lab, your assignment is eligible for up to an 'A-'. Continue on a bit further for some decryption fun (and as a side effect, a potentially higher grade).

(c) bestDecryption(self). This property method should automatically decrypt interesting messages, written in English. Since encrypted messages and incorrectly decrypted messages are unlikely to include many words, we can use the wordCount property (a property of all of these classes) to check for possible plaintext. This method's algorithm simply computes all the 26 possible decryptions (Plaintext objects) and returns the Plaintext object whose text property mentions the most words. If multiple decryptions are equally good, return any of them.

```python
>>> plain = Ciphertext('jgnnq').bestDecryption
>>> plain.text
'hello'
```
Remark. Wow! We’ve built three classes—Message, Plaintext, and Ciphertext—that all share attributes and bits of code. The base class, Message provides the basic functionality for all three. All of these classes include text and wordCount properties, but we only wrote those methods once. That’s the power of inheritance. The derived classes differ, of course: Plaintext has an encrypted property, while Ciphertext has a decrypt method. They are specialized forms of Message. Now on to the fun stuff!

Task 5. Decrypting a Mystery Message.

There is an encrypted message contained in the story.txt file. Use the loadStory() helper method and demonstrate how your classes work by decrypting and printing the associated plaintext. Place that code in the if __name__ == '__main__': after the doctest testing code. If you can read the story, your code is working successfully!

Submitting your work. When you’re finished, stage, commit, and push your work to the server as you did in previous labs. Remember that you must certify that your work is your own, by typing out the Honor Code statement in the honorcode.txt file, committing and pushing it along with your work.

Late days. You are allowed a total of 3 late days over the semester, with at most 2 late days towards any one lab. You must request a late day in advance on the form, here: http://bit.ly/s20late.

Grading Guidelines. Both functionality and programming style are important when writing code, just as both the content and the writing style are important when writing an essay. In this program, some of the specific functional requirements we will test for include:

- Your class definition must pass all the doctests. Add a doctest to each method as described in the starter code.
- Subclasses should make use of the methods and attributes available via their superclass. Avoid repetitive code that duplicates what the superclass does!
- Properties and setters exist as public methods accessing and modifying private attributes and methods (i.e., avoiding “underscore guilt”).
- If you have everything else working except bestDecryption(self), your lab is eligible for up to an ‘A-‘.
- Your if __name__ == '__main__': should display the decrypted story.
- Just like previous labs, we require that the functions defined in cipher follow our specifications. Do not modify the method names, their parameters, nor what is returned. The examples above in This week’s tasks provide details about these constraints.

Stylistically, we expect to see programs that exhibit the following: meaningful names used in declarations, informative comments, docstrings for all methods and at the top-level, good and consistent formatting, and good choice of Python commands. There is some subjectivity to what makes good style, but the basic goal is to make your ideas as clear and easy to follow as possible.

Grading scale. Programming labs will be graded on the following scale:
<table>
<thead>
<tr>
<th>Grade</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A+</td>
<td>An absolutely fantastic submission of the sort that will only come along a few times during the semester.</td>
</tr>
<tr>
<td>A</td>
<td>A submission that exceeds our standard expectations for the assignment. The program must reflect additional work beyond the requirements or get the job done in a particularly elegant way.</td>
</tr>
<tr>
<td>A-</td>
<td>A submission that satisfies all the requirements for the assignment – a job well done.</td>
</tr>
<tr>
<td>B+</td>
<td>Submission meets the requirements for the assignment, possibly with a few small problems.</td>
</tr>
<tr>
<td>B</td>
<td>A submission that has problems serious enough to fall short of the requirements for the assignment.</td>
</tr>
<tr>
<td>C</td>
<td>A submission that has extremely serious problems, but nonetheless shows some effort &amp; understanding.</td>
</tr>
<tr>
<td>D</td>
<td>A submission that shows little effort and does not represent passing work.</td>
</tr>
</tbody>
</table>

Acknowledgment. This lab assignment has been adapted from MIT's Introduction to Computer Science and Programming in Python Fall 2016 course materials.

*