Computer Science CS134 (Fall 2020)
Daniel Aalberts, Duane Bailey, & Molly Feldman
Laboratory 8
Simple Ciphers (due Thursday at 5pm)

Objective. To use inheritance to quickly construct a new character substitution cipher.

In this lab, we'll experiment with the use of *inheritance* to help us build classes that support a variety of *cipher* mechanisms. A cipher is an approach to transforming the characters of a readable *plaintext* message into an *encoded* message. It's necessary, of course, for a successful cipher technique to provide a *decoding* method that allows the recipient to recover the plaintext.

The ciphers we'll implement this week range from the simple to the complex, but every cipher technique, `coder`, supports two main methods: `coder.encode(sourceString)` and `coder.decode(sourceString)`. We'll provide these in a *base class*, `Cipher`, that we will extend several new classes that support solid cipher implementations. This approach is important for two reasons. First, utilities that make use of ciphers, for example tools for reading and writing encoded files, will be more flexible if all our cipher techniques support the same methods. We can quickly substitute more or less secure approaches by a simple substitution of *subclasses* of `Cipher` that are used in a consistent manner. Second, when we realize that several classes will share the same *interface* with the outside world, there is a promise that code can be shared. Thoughtful sharing of code through a central `Cipher` implementation will make new implementations very simple.

Code Review. Let's download and review this week's code.

Download the repository in the usual manner, using your name instead of `22xyz3`:

```
git clone https://evolene.cs.williams.edu/cs134-labs/22xyz3/lab08.git ~/cs134/lab08
```

We'll focus, mainly, on one file in this repository: `cipher.py`. Browse through this code as we discuss it below.

In this module we have defined a new class, `Cipher`. This class is not meant to be directly used. Instead, it describes the interface common to all the cipher mechanisms we'll implement. Again: the interface is simply the methods that we promise our users will be available. This includes, among others, the methods we expect them to use (like `encode(sourceString)` and `decode(sourceString)`). The `Cipher` class is also a convenient place for us to find public and private methods we would like to write once and use in many subclasses.

Here are the highlights of the `Cipher` class:

- An empty `__slots__` variable. Users will not able to hang unwanted attributes off `Cipher` or any of its subclasses.
- A simple `__init__(self)` method. This basic class does not have any state, so its initializer does nothing. Later, if we want to share implementations of state common to all our ciphers, we can add them as state variables (in `__slots__`) and initialize them here.
- The `encode(self,sourceString)` method. Ideally, this method will accept a plaintext string (`sourceString`) and return its encoded equivalent. The implementation here simply returns the string, unmodified.

- The `decode(self,sourceString)` method. Similarly, this method would normally reverse the encoding, returning the plaintext. Again, this implementation does nothing.
- The `_a2i(self,c)` method. This converts an alphabet letter `c` to its alphabet code (in `range(26)`). For example, calling `self._a2i('d')` (or `self._a2i('D')`) within a cipher returns the integer 3.
- The methods `_i2a(self,i)` (and `_i2A(self,i)`) reverse this process, converting alphabet codes to lowercase (and uppercase) letters.
- The `_rotate(self,c,n)` "rotates" the letter `c` forward through (and around) the alphabet by `n` letters. For example, `self._rotate('I',-1)` returns `'H'`. It does not modify non-letters.
- The `_normalize(self,sourceString)` method. This method extracts the alphabetic characters from `sourceString` and converts them to uppercase. Many of our ciphers do not extend to non-letters, so spacing and punctuation make the cipher more prone to attack. We'll use `_normalize` to generate a "canonical form" for message text.

As we gain more experience, we may find there are other methods that could be shared among cipher implementations; we'll place them here. We may also find improvements in the public interface. If that happens, we'll enforce those changes here.

We've also included a `CaesarCipher` class here, as well. It's declared:

```
class CaesarCipher(Cipher):
    ...
```

This is how we indicate that `CaesarCipher` is *an extension* or *a subtype* of `Cipher`. Because of that relationship, all the features of `Cipher` are *inherited* by `CaesarCipher`. If we wish, we can reimplement or *override* implementations of methods that must be specialized for this particular `Cipher`.

Note these specializations:

- The `CaesarCipher` declares a state variable, `self._n`. This keeps track of the degree of forward rotation used for encoding. (Decoding, obviously shifts backward by negating `self._n`.)
- This state variable is initialized in `__init__(self,n)`, with an integer passed in the constructor. Before the initializer does anything, it calls its super-class's initializer. This is accomplished with:

  ```
  super().__init__()
  ```

  This ensures that any part of the state that `Cipher` is responsible for is initialized before we initialize any state in the subclass. In most cases, we don't have to specify an initializer; the default is to inherit and call the `Cipher` initializer. When we extend state, we must be sure to explicitly call the superclass initializer.
- It provides `encode(self,sourceString)` and `decode(self,sourceString)` methods that apply the same rotation to every element of the *normalized* plaintext. The removal of spaces and punctuation greatly improves its security.
- There is a `__repr__(self)` method that describes how this `CaesarCipher` can be constructed.

We can exercise our CaesarCipher from within interactive Python as follows:

```
>>> from cipher import CaesarCipher
>>> coder = CaesarCipher(1)
>>> coder.encode('H.A.L.')
'IBM'
>>> code = open('story.txt').read()
>>> code[:26]
'GJBDGHPNHIWPILWTCEGTHXSTCI'
>>> coder2 = CaesarCipher(15)
>>> coder2.decode(code[:26])
'RUMORSAYSTHATWHENPRESIDENT'
```

Our work this week will be the development of new ciphers which extend Cipher and its subclasses.

Required Tasks. This week, we would like you to create and test two new cipher classes, for a full 10 points.

1. In cipher.py build a new class, Rot13(), that is a subclass of CaesarCipher(n).
   (a) This class's parameterless initializer does one thing: it calls its superclass initializer with a fixed value of n, 13.
   (b) Since Rot13 is not meant to be secure, override the _normalize(sourceString) method so that it *does not* remove punctuation or change case. This allows us to preserve the punctuation in cases where it's important:

   ```
   >>> code = open('reform.txt').read()[:46]
   >>> coder = Rot13()
   >>> plain = coder.decode(code)
   'A Plan for the Improvement of English Spelling'
   >>> coder.encode(plain)
   'N Cyna sbe gur Vzcebirzrag bs Ratyvfu Fcryyvat'
   ```

   (c) Demonstrate the functionality of Rot13 by writing some new simple doc-tests on its initializer.
   (d) Write an appropriate __repr__(self) method.
   (e) Notice how you *inherited* encode and decode. You need not write these methods because they were already written in CaesarCipher.

2. In cipher.py, build a new subclass of Cipher, Vigenere.
   (a) This cipher's initializer takes a string, key, normalized to all uppercase letters. This key is saved in a state variable and serves to determine the shift amount during encoding and decoding.
   (b) Provide an encode(self,sourceString) method. Normalize sourceString. At position i of the normalized plaintext, rotate the letter using letter i of the key (repeating the key, as necessary): the *alphabet code* associated with the key letter is interpreted as the amount to rotate the plaintext letter. Using 'WILLIAMS' as the key, the first letter of 'PURPLECOWS' is shifted 22 (the alphabet code of W) spaces in the alphabet to L. Any A in the key causes the corresponding letter to not shift (i.e., stay the same):

plaintext	P	U	R	P	L	E	C	O	W	S
key	W	I	L	L	I	A	M	S	W	I
shift	22	8	11	11	8	0	12	18	22	8
encoded	L	C	C	A	T	E	O	G	S	A

(c) Override the `decode(self,sourceString)` method to decode your cipher.

(d) Override the `__repr__(self)` method.

(e) Thoroughly test your class. At the very least, it should act as follows:

```
>>> vCoder = Vigenere('Williams')
>>> vCoder.encode('purple cows')
'LCCATEOGSA'
>>> vCoder.decode('lccateogsa')
'PURPLECOWS'
>>> code = open('cia.txt').read()[:51]
>>> coder = Vigenere('SANBORN')
>>> code
'KAACCIAKKEZDKBKIFBGTHDPGVFVYGCNUSUBFTUFUIBMNQTCWPAA'
>>> coder.decode(code)
'SANBORNSKRYPTOSISASCULPTURELOCATEDONTHEGROUNDSOFCIA'
>>> coder
Vigenere('SANBORN')
```

3. Review and document your code, adding appropriate symbols to the `__all__` method.

4. Sign the honor code. Add, commit, and push your changes to `cipher.py`.

⋆

The following is a fun little project with another coded text to decode. This part of the lab is for independent investigation and will not be graded.

A really tough cipher. The Enigma Machine[1] was a physical coding machine that generated very complex substitution ciphers. The operators would set a set of three rotors to their "zero" position (indicated by a 3-letter key-of-the-day). They would then press a letter on the keyboard. The signal would get substituted or *scrambled* as they went through the rotors. Eventually, the signal reached a display where the encoded character was read. The internal racheting mechanism would then spin the rotors one notch, effectively changing the encoding of the next character pressed. It was a devastating device.

In the `scrambler` module, we've built a `scrambler(c,spin,key)` function that simulates the action of the Enigma's rotors on character `c`, after `spin` characters from the beginning of the message to be encoded with `key`. You can play with the scrambler by hand:

```
>>> scrambler('A',0,start='EPH')
'H'
```

What is amazing is the Enigma is a *mirror encoder*. The same device, set up in the same state, fed the encoded text will regenerate the plaintext!

```
>>> scrambler('H',0,start='EPH')
'A'
```

To build your own Enigma-like machine, here are the steps:

1. Declare a new subclass of `Cipher` called `Enigma`.
2. You'll need two state variables that keep track of the key-of-the-day, and the current value of `spin`. Keeping the value of `spin` as state outside of the `encode` method will allow you to encode (or decode) a long message using several calls to `encode`. You may find it useful to have a `reset(self)` method that allows you to reset `spin` before each batch of encoding or decoding.
3. The `encode` method (`decode` is identical) takes a normalized string. It then calls the `scrambler` with each character of the plaintext, `spin`, and the `key`. After each operation `spin` is incremented. Here is a basic test of functionality:

   ```
   >>> coder = Enigma('TUR')
   >>> code = open('turing.txt').read()[:35]
   >>> code
   'EFBFKPRLJNLFGYFEPMQZXPFMWSKKLROTIHG'
   >>> coder.decode(code)
   'TURINGWASTHEFATHEROFCOMPUTERSCIENCE'
   ```

   Can you translate the rest of the message?

Congratulations!

[1] https://bletchleypark.org.uk/our-story/the-challenge/enigma