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 supports two main methods: encode(sourceString) and 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. Download the repository in the usual manner, using your name instead of 22xyz3:

```bash
git clone https://evolene.cs.williams.edu/cs134-labs/22xyz3/lab07.git ~/cs134/lab07
```

where your CS username replaces 22xyz3. 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. We also use it to keep track of a common attribute of ciphers, the alphabet used as the basis for encoding and decoding.

Here are the highlights of the Cipher class:

- A __slots__ variable containing the alphabet. 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 do much: it simply gives an initial value to the alphabet attribute. Later, if we want to share other attributes 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 \texttt{a2i(self,c)} method. This converts an alphabet letter \(c\) to its alphabet code (according to \texttt{alphabet}, in range(26)). For example, calling \texttt{self.a2i('d')} (or \texttt{self.a2i('D')}) within a cipher returns the integer 3 using a standard \texttt{alphabet}.

The methods \texttt{i2a(self,i)} (and \texttt{i2A(self,i)}) reverse this process, converting alphabet codes to lowercase (and uppercase) letters.

The \texttt{rotate(self,c,n)} "rotates" the letter \(c\) forward through (and around) \texttt{alphabet} by \(n\) letters. For example, \texttt{self.rotate('I',-1)} returns 'H' in the standard alphabet. It does nothing to non-letters.

The \texttt{normalize(self,sourceString)} method. This method prepares text for encoding or decoding. By default, this method simply returns the source string. In other cases, this might extract the alphabetic characters from \texttt{sourceString} and convert them to uppercase. Our ciphers do not do anything to non-letters, so spacing and punctuation make the encoded text more prone to attack. We’ll use \texttt{normalize} to generate a “canonical form” for cipher-related strings.

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

We’ve also included a \texttt{RotCipher} class here, as well. This class implements a simple cipher (sometimes called a \textit{Caesar cipher}), where letters are encoded by rotating them through the alphabet. It’s declared:

```
class RotCipher(Cipher):
...
```

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

Note these specializations:

- The \texttt{RotCipher} declares an additional state variable, \texttt{self.n}. This keeps track of the degree of forward rotation used for encoding. (Decoding, obviously shifts backward by negating \texttt{self.n}.)

- This \texttt{n} attribute is initialized in \texttt{__init__(self, n)}, with an integer passed in the initializer. Before the initializer does anything, it calls its superclass’s initializer. This is accomplished with:

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

This ensures that any part of the state that \texttt{Cipher} is responsible for (for example, the \texttt{alphabet} attribute) is initialized before we initialize any state in the subclass. In most cases, we don’t even have to write an initializer: the default is to inherit and call the initializer in the superclass. When we add attributes to the state (as we do in \texttt{RotCipher}) we \textit{must} be sure to explicitly call the superclass initializer before we initialize these new state variables.

- It specifies new implementations of the \texttt{encode(self,sourceString)} and \texttt{decode(self,sourceString)} methods that apply the same rotation to every letter of the \textit{normalized} plaintext.
There is a \_\_repr\_\_(self) method that describes how this RotCipher can be constructed. Passing this string to eval effectively clones the RotCipher.

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

```python
>>> from cipher import RotCipher
>>> coder = RotCipher(1)
>>> coder.encode('H.A.L.')
'I.B.M.'
>>> code = open('story.txt').read()
>>> code[:30]
'Gjbdg hpnh iwpi lwtc Egthxstci'
>>> coder2 = RotCipher(15)
>>> coder2.decode(code[:30])
'Rumor says that when President'
```

Our work this week will be the development of new classes that extend Cipher and RotCipher.

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, TajCipher()\[1\] that is a subclass of RotCipher(n). A TajCipher is a RotCipher that makes use of an alternate alphabet. Typically, we provide a pangram\[2\] as a mnemonic for specifying an alternative order for the letters of the alphabet. This alternative alphabet makes it possible to increase the difficulty of cracking messages encoded with Caesar-style ciphers.

   (a) Write the \_distill\_(self,pangram) method. This method takes a phrase—typically a pangram—and returns a lowercase string of letters in the order they first appear in the phrase. We expect the result will be exactly 26 characters long. For example, if coder is a TajCipher

       coder._distill('Avi, Tally, and Jerry go quick when we fumble six pizzas!')

   would return the alternative alphabet order 'avitlyndjergoquckwhfmbspzx'.

   (b) Write the TajCipher's initializer. It takes two parameters: n and a pangram. The n is used to initialize the RotCipher superclass. The distilled pangram is used as an alternative value for the attribute _alphabet inherited from Cipher. If no pangram is specified, it should default to the order suggested above.

       >>> coder = TajCipher(0)
       >>> len(coder._alphabet)
       26
       >>> coder = TajCipher(0, 'The quick brown fox jumps over the lazy dog.')
       >>> coder._alphabet
       'thequickbrownfxjmpsylazydg'

   (c) Write an appropriate \_\_repr\_\_(self) method. Make sure that, if the string were passed to eval, it would generate a TajCipher cipher with exactly the same behavior:

---

1 Named after the cipher's inspiration, Tally, Avi, and Jerry.
2 Recall, a pangram is a phrase that makes use of every letter in the alphabet.
>>> coder = eval(repr(TajCipher(5,'The five boxing wizards jump quickly.')))
>>> coder
TajCipher(5, 'thefivboxngwzardsjumpqckly')

(d) We're done writing code for this class. You inherited encode and decode. You did not need to write these methods because they were already written in RotCipher. You should be able to perform much more serious testing:

>>> code = open('reform.txt').read()[:46]
>>> coder = TajCipher(13)
>>> plain = coder.decode(code)
>>> plain
'A Plan for the Improvement of English Spelling'
>>> coder.encode(plain)
'Q Gwqf nzx kys Cdgxzusdfs fzn Sfpwcey Egswwcfp'

(e) Demonstrate the functionality of TajCipher by writing some new simple doc-tests on its initializer. (Hint: Can a TajCipher act like a RotCipher?)

2. In cipher.py, build a new subclass of Cipher, Vigenere. Vigenere ciphers are extremely difficult to crack.

(a) Write a method, _normalize(self, sourceString), that overrides the do-nothing method from Cipher. This method extracts the alphabetic characters from sourceString and converts them to uppercase. You can probably improve on this implementation:

```python
def _normalize(self, sourceString):
    """Return an uppercase string of the letters from sourceString."""
    result = []
    for character in sourceString:
        if character.isalpha():
            result = result + [character.upper()]
    return ''.join(result)
```

(b) The Vigenere initializer takes a string, key, normalized to just uppercase letters. This key is saved in a new attribute and serves to determine the shift amount during encoding and decoding. At this point you can test Vigenere in a manner similar to the following:

```python
>>> Vigenere('Williams!!!')._key
'WILLIAMS'
```
If this test works, move on.

(c) Override the __repr__(self) method. Again, we expect the this method will return a string describing how to build an equivalent Vigenere:

```python
>>> coder = eval(repr(Vigenere('Williams!!!')))  
>>> coder
Vigenere('WILLIAMS')
```
(d) Write 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 default _alphabet to L. Any A in the key causes the corresponding letter to not shift (i.e., stay the same):

<table>
<thead>
<tr>
<th>plaintext</th>
<th>P</th>
<th>U</th>
<th>R</th>
<th>P</th>
<th>L</th>
<th>E</th>
<th>C</th>
<th>O</th>
<th>W</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>W</td>
<td>I</td>
<td>L</td>
<td>L</td>
<td>I</td>
<td>A</td>
<td>M</td>
<td>S</td>
<td>W</td>
<td>I</td>
</tr>
<tr>
<td>shift</td>
<td>22</td>
<td>8</td>
<td>11</td>
<td>11</td>
<td>8</td>
<td>0</td>
<td>12</td>
<td>18</td>
<td>22</td>
<td>8</td>
</tr>
<tr>
<td>encoded</td>
<td>L</td>
<td>C</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>E</td>
<td>O</td>
<td>G</td>
<td>S</td>
<td>A</td>
</tr>
</tbody>
</table>

We expect, in both `encode` (and below, in `decode`) you'll use the inherited helper methods `.a2i`, `_i2A`, and `_rotate`.

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

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

```python
>>> vCoder = Vigenere('Williams')
>>> vCoder.encode('purple cows')
'LCCATEOGSA'
>>> vCoder.decode('lccateogsa')
'PURPLECOWS'
>>> code = open('cia.txt').read()[:51]
>>> coder = Vigenere('SANBORN')
>>> code
'KAACCIAKKEZDKBKIFBGTHDPGVYGCNUSUBFTUFUIMNQTCWPAA'
>>> coder.decode(code)
'SANBORNKRYPTOSISASCULPTURELOCATEDONTHEGROUNDSOFcia'
>>> 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\textsuperscript{[3]} 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 ratcheting 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\( (c,\text{spin},\text{key}) \) function that simulates the action of the Enigma's rotors on character \( c \), after \( \text{spin} \) characters from the beginning of the message to be encoded with \( \text{key} \). 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\( (\text{self}) \) 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
'EFBFKPRLLNLFGYFEPMQZXPFMWSKKLROTIHG'
>>> coder.decode(code)
'TURINGWASTHEFATHEROFCOMPUTERSCIENCE'
```

Can you translate the rest of the message?

*Congratulations!*  

\textsuperscript{[3]}https://bletchleypark.org.uk/our-story/the-challenge/enigma