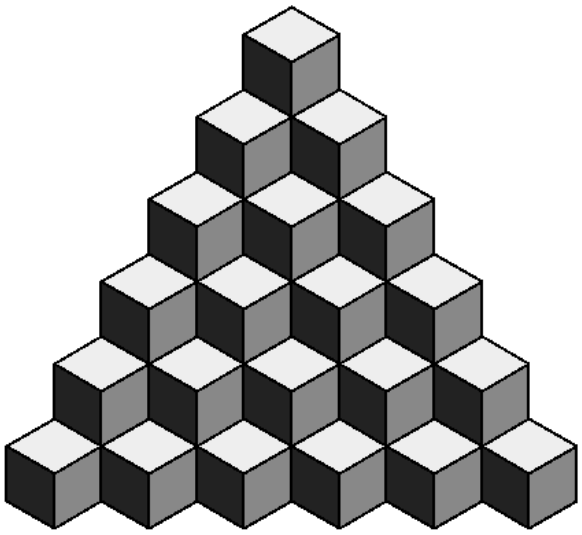
You have three questions to complete this week: one concerning using `GPolygon` and `GCompound` and then two dealing with string manipulations. Do not forget to adjust the README to indicate you have completed the assignment before your final commit!

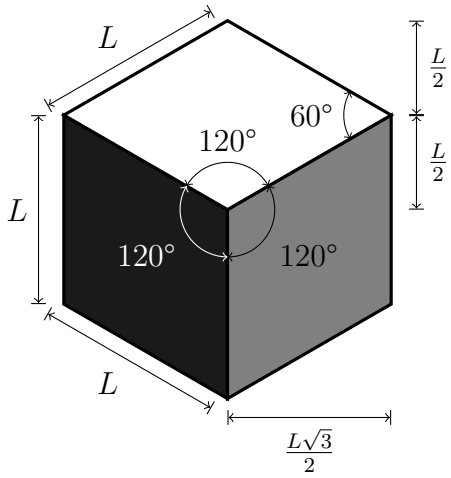
Get Assignment link: https://classroom.github.com/a/AiKorJC6

1. There is a classic optical illusion that looks like:



with the illusion arising from the fact that it is possible to see the white surfaces as either the tops or the bottoms of cubes stacked to form a pyramid. We'd like here to create a "paintbrush" of sorts that would allow you to create pyramids like this or any other geometric creation you could imagine.

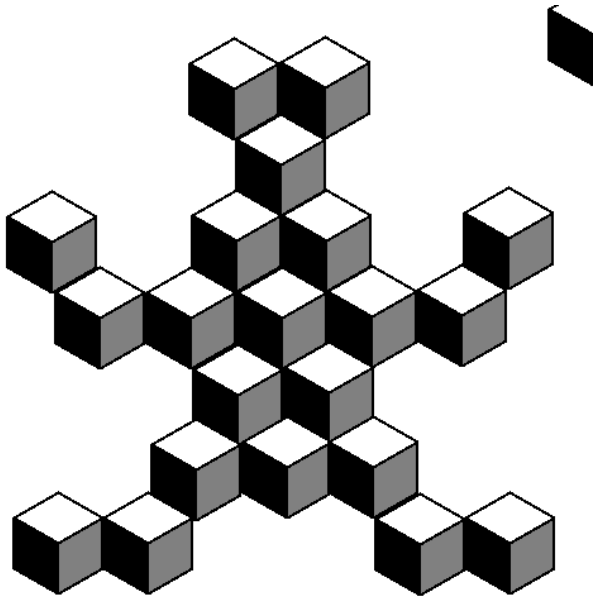
To do so, you should first define a function `create_cube` to draw a single cube, which will be returned as a `GCompound` object. Each cube GCompound is comprised of 3 `GPolygon`s, each with a differently shaded face:



As you are drawing these, it is helpful to know that each edge you draw is always the same length (shown here as L), and should be 40 pixels. For the colors, you should actually set the fill colors here, so that you keep the black outline of the poylgon. I have also added a few extra geometric bits of information to the above image to potentially help you place the vertices correctly. Remember that both GCompounds *and* GPolygons use their own coordinate systems! *When you return the final cube GCompound, it should have the origin placed at the center meeting point of the three polygons.*

Once you have your function creating a cube GCompound, and have ensured that it can be drawn to the screen and looks correct, add one cube to the window. Then set up an event listener and callback function so that whenever the mouse moves, the cube is moved to the same location as the mouse. Moving your mouse around the screen should now cause the cube to follow it. This will serve as our "paintbrush" cursor.

To achieve our "painting" effect, we will add one more mouse event (and corresponding callback function) when you `"mousedown"`. Whenever you press the mouse down, a new cube should be created and added to the screen wherever your cursor currently is. Now you can draw whatever geometric shapes you like! One of my examples looked like below. The half cube in the upper right was where my cursor was when I left the window to take the screenshot.



2. Many people in English-speaking countries have played the Pig Latin game at some point in their lives. There are other invented "languages" in which words are created using some simple transformation of English. One such language is called *Obenglobish*, in which words are created by adding the letters *ob* before the vowels (*a,e,i,o* and *u*) in an English word. For example, under this rule, the word *english* gets the letters *ob* added before the *e* and the *i*, to form *obenglobish*, which is how the language got its name.

   In official Obenglobish, the `ob` characters are only add before vowels that are pronounced, which means that a word like *game* would become *gobame* rather than *gobamobe* since the final *e* is silent. While it is impossible to implement this rule perfectly, you can do a pretty good job by adopting the rule that *ob* should be added before every vowel in the word **except when**:

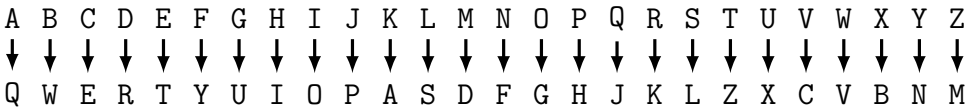
   - the vowel follows another vowel
   - the vowel is an *e* occurring at the end of a word

   Write a function `to_obenglobish` that takes an English word as an argument and returns its Obenglobish equivalent, using the translation rule given above. For example, your function should be able to output something similar to below:

```
>>> print(to_obenglobish("english"))
obenglobish
>>> print(to_obenglobish("gooiest"))
gobooiest
>>> print(to_obenglobish("amaze"))
obamobaze
>>> print(to_obenglobish("rot"))
robot
```

   Don't forget about decomposition! I found it very useful in this problem to write predicate functions which take care of checking the above special conditions.

3. A letter-substitution cipher operates by consistently replacing a letter with a corresponding letter from a given key. The key in a letter-substitution cipher is a 26-character string that shows the enciphered counterpart of each of the 26 letters of the alphabet. For example, if both communicating parties chose `"QWERTYUIOPASDFGHJKLZXCVBNM"` as the key (unimaginatively chosen by typing the letter keys on the keyboard in order), then that key would correspond to the following mapping:



Your task here is to write a function `encrypt` which takes two arguments: the message to encrypt and the 26-character key, and which returns the encrypted message. Any capitalized letters should be capitalized in the encrypted message, and lower-case letters should be lower-case in the encrypted message. Any characters which are not letters should be unchanged in the encrypted message. For example, with the above key, then you should be able to get something like:

```
>>> KEY = "QWERTYUIOPASDFGHJKLZXCVBNM"
>>> print(encrypt("Squeamish Ossifrage", KEY))
Ljxtqdoli Glloykqut
```

As a fun aside, the words *squeamish ossifrage* were part of the solution to a cryptographic puzzle published in *Scientific American*. The puzzle was developed by Ron Rivest, Adi Shamir, and Leonard Adleman, who invented the widely used RSA encryption algorithm, named from the first letters of their surnames. As some other examples, using the same key as above, you should be able to get the below output:

```
>>> print(encrypt("ABC - 123", KEY))
QWE - 123
>>> print(encrypt("Isn't this great?"), KEY)
Olf'z ziol uktqz?
```

Note that, although the key is comprised of entirely capitalized letters in the example here, it does not *need* to be. So your program should be able to work properly for keys that are either capitalized or lower-case.