Computer Science CS134C (Fall 2018)
Duane A. Bailey
Laboratory 7
Building Image Filters (due Thursday/Friday)

Objective. To learn how to use classes to reduce code complexity.

In this lab, we'll investigate how we might write code to transform images. For example, we'll think about how to convert color images to monochrome images using a grayscale filter. Our work here actually duplicates efforts in the *Python Image Library* (PIL) but will be a bit less efficient. Still, our approach to building special-purpose image filters parallels techniques we'll commonly use to build up toolkits to solve other big data problems.

Before we get started, we're going to need to install the PIL module. We first activate our virtual environment and then install `pillow`, the python package that includes PIL:

```
-> cd ~/cs134
-> source bin/activate
(cs134) -> pip install pillow
...
Successfully installed pillow-5.3.0
(cs134) ->
```

Now, make sure you clone your `lab7` repository. Recall that if you are `22xyz9`, you type:

```
git clone ssh://22xyz9@davey.cs.williams.edu/~cs134/22xyz9/lab7.git ~/cs134/lab7
```

The RGB Colorspace Model.

This lab depends heavily on the red-green-blue (RGB) colorspace model. This models colors that occur as the result of mixing three colored lights with varying intensity. For this lab, we'll encode the intensities as integer values between 0 and 1. When the intensity is 0, that light does not contribute to the final mixture while an intensity of 1 indicates that light is on full. We'll think of a colors, then, as being represented by 3-tuples (viz. `(r,g,b)`) of integers between 0 and 1. The color black is represented by `(0,0,0)` and white is `(1,1,1)`. Red would be represented by `(1,0,0)`, and dark red might be `(0.25,0,0)`. All monochrome values (shades of gray) have equal amounts of red, green, and blue.

The HSV Colorspace Model.

We may also wish to make use of the HSV colorspace model which, more appropriately models the mixing of paints. The components are hue (H), saturation (S), and value (V). The hue is specified as an *angle* measured in degrees. Zero degrees is red, 120 is green, and 240 is blue (in the same order as R-G-B). Hues are "mixed" by averaging known hue values. Thus, yellow is half way between red and green at 60 degrees. Notice that yellow's complement is blue, 180 degrees away. Similarly, we find the cyan is 180 degrees and magenta is 300.

Now, imagine you have a paint can filled with paint colored with a hue. The saturation is a value between 0 and 1 that indicates *what percentage of the original hue-colored paint you keep, topping the rest off with white*. As saturation decreases, the resulting color becomes lighter, less saturated.

Finally, take the saturation-corrected can of paint and interpret the value as *the percentage of that paint you keep, topping the rest off with black*. As value decreases, the resulting color becomes darker.

The Color Class

In the filters module, you'll find the Color object. Everything in this lab uses this class to describe colors of pixels. You can construct colors using either the RGB or HSV color model. Given a color, you can pull out the RGB or HSV definitions with the rgb() and hsv() methods, respectively.

```
>>> from filter import Color
>>> yellow = Color((1,1,0))   # construct a color from RGB components
>>> yellow.hsv()
(60.0, 1.0, 1.0)
>>> yellow = Color((60,0,0))  # construct a color from HSV components
>>> yellow.rgb()
(1.0, 1.0, 0.0)
```

You can also pull out individual components with red(), green(), blue(), hue(), saturation(), and value().

The Filter Classes.

In the file filter.py you'll find a number of classes that we have written that describe *filters*. These are objects that, when they're constructed can be used to perform a transformation on the pixels of an image. For example, you can use the GrayFilter, which transforms the color pixels (the *before* pixels) of an image into gray pixels (the *after* pixels). If you're finished with modifying the image, you can ask for the result of this process as a PIL Image and then save it to disk.

```
from PIL import Image
flowers = Image.open('Irises.png')
filteredImage = GrayFilter(flowers)
grayIrises = filteredImage.image()
grayIrises.save('GrayIrises.png')
```

Once saved to disk, you can type

```
(cs134) -> open GrayIrises.png
```

to preview the image.

Tasks to be done.

You'll find a file, `lab7.py` in your repository. We'd like you to add a few new filters to that module. You'll find a class, `Template`, that you can cut-and-paste as a start. Typically, a new filter needs to have its name changed (from `Template`) and you need to modify the `after(x,y)` method to describe how you'll transform the `before` pixels. Here's what we need from you:

1. Write two new filters: `LighterFilter` and `DarkerFilter`. These bring the RGB values closer to white and black, respectively. Test these out, using PNG images of your choice. Make sure your filters can be composed; e.g. `GrayFilter(LighterFilter(flowers))`.

2. Write a filter that remaps the colors of one of Van Gogh's images: *Bedroom*, *Sunflowers*, or *Roses*. We have the conservator's expectation of *Bedroom* as *BedroomExpected*, which increases the use of red. In *Sunflowers*, we hope to make the muddy brown-green more yellow, while in *Roses*, we hope to make the lightest areas more pink. `FadingColors.pdf` is a technical discussion of the chemistry.

3. If you wish, write a fourth filter of your choice (credit varies based on effort). Here are some ideas:

   (a) Write a filter that keeps only pixels that are 'close to' a target color, while other pixels become white. You'll have to extend the template class to include a slot that remembers the target color. You'll also have to think about a distance metric.

   (b) Write a filter that blurs an image. This filter computes a weighted average of RGB pixels in a neighborhood. This is called a *kernel*. A kernel for blurring is

   ```
   1 1 1 1 1
   1 0 0 0 1
   1 0(0)0 1
   1 0 0 0 1
   1 1 1 1 1
   ```

   Imagine the pixel you're computing is located at the center of the kernel (the parenthesized 0), and compute the weighted sum of all the pixels relative to this center pixel. For example, the pixel at $(x-2, y-2)$ is weighted 1, but the pixel at $(x-1, y-1)$ is weighted 0. After computing the sum, the blurred pixel value is this sum divided by 16 (the sum of all the weights in the kernel). You'll need to think about how to multiply a color by a weight and how to sum them, of course. You'll also have to think about how to represent the kernel, itself.

   You might think about how this filter works.

   (c) Write an edge-enhancement filter. It has the kernel

   ```
   -1   -1   -1
   -1  (10)  -1
   -1   -1   -1
   ```

   The average is computed by dividing by 2. This filter is also interesting to think about.

4. Document your efforts. Make sure your module, its classes, and its methods all have doc comments.

5. Push your repository.

⋆