COSC 101: Introduction to Computing I
Homework 4
Fall 2013

In this homework you will write a series of programs to perform different manipulations of digital images. The goal of this homework is for you to get practice with `for` loops, nested `for` loops, and conditional statements.

This homework is **due on Wednesday, October, 2, 2013 11:55pm**. As before, please use good variable names and include a few descriptive comments (e.g., at the very least, label which for loops do which transformation!).

1 Getting started

- If you have not already, read the background information on image transformations. It is included in sections 5 and 6 of this document. The same information is included in the lab 4, so if you have already read it, you can skip this part.
- If you have never seen rutabaga paired with garlic and bok choy, then you probably need to re-read the background information before continuing.
- Open the demo program, called `image_demo.py`, in IDLE and make sure it runs properly. *Do not attempt the tasks below until you can run the demo programs successfully.*
- Open `hw4.py` in IDLE. You should write all of your programs in this file. If you choose to write separate programs for each of the problems below, be sure that each file is saved in the same folder as `image.py` and the other files we have provided.

2 Your task

Your task is to complete the following seven image transformations. **Important**: your final program should have **one** call to `display_images` at the *end* of your program that displays all **seven** images that you create in this lab.

A challenge problem is mentioned in Section 4.

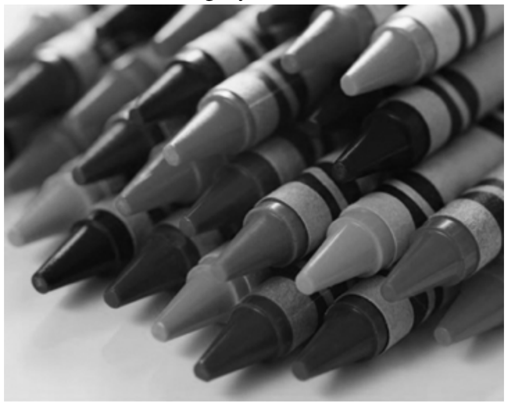
3 Pixel-by-pixel transformations

The following image transformations alter the image by applying the same transformation to each pixel in the image. You can solve each one by writing a pair of nested loops similar to what is done in red filter example program described in Section 6.3. The main difference is in which color values are examined and how the colors are changed.

1. **Grayscale**. Shades of gray have the same red, green and blue value. To convert an image to grayscale, you want to set the red, green and blue values all to the average value of the three channels of the original pixel.

original

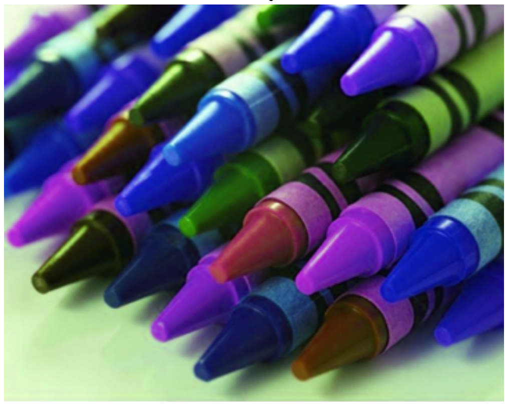
grayscale



2. **Color cycle**. Set the blue value to the original green value, the green to the original red, and the red to the original blue.

original

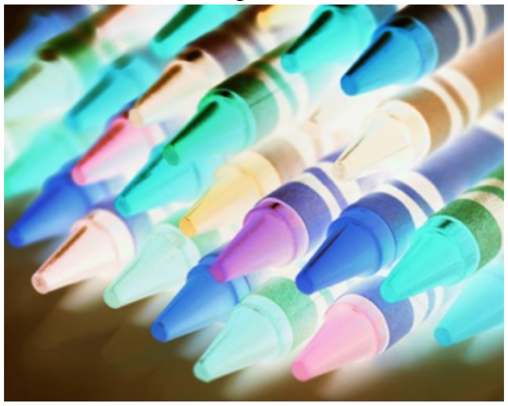
color cycle



3. **Negatives**. The negative of an image is creating by inverting each color channel. So if the red value of a pixel were 255, it should become 0. If it were 254, it should become 1, and so on, down to 0, which should become 255. Similarly for green and blue.

original

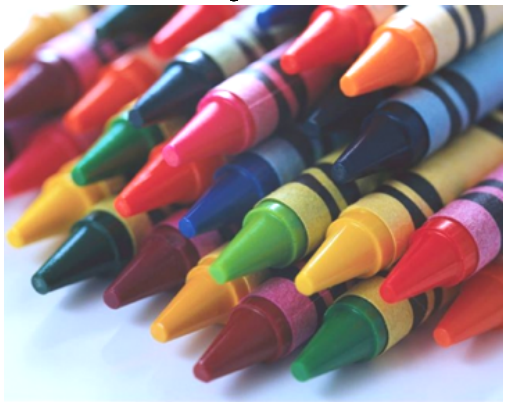
negative



4. **Brightness**. Prompt the user for an integer change to the brightness. This change may be positive or negative. Adjust the color values by the entered amount. Just be sure no values go over 255 or below 0. In this example, the brightness was increased by 40.

original

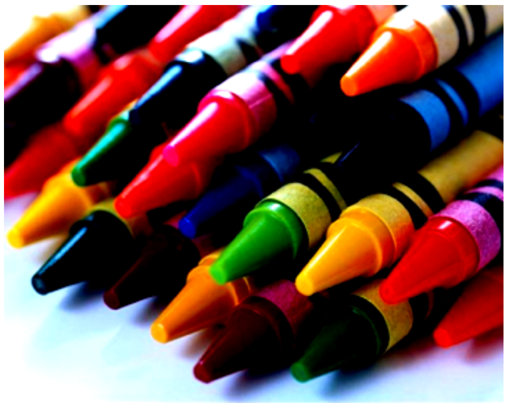
brightness



5. **Increase Contrast**. When increasing the contrast, color values at 128 should be unchanged. For any other value x, the difference between x and 128 should be scaled a factor of 2. For example, a color value of 129 (1 above 128) would become 130 (2 above 128). 125 (3 below 128) would become 122 (6 below 128). Just remember that you'll need to stay between 0 and 255.

original

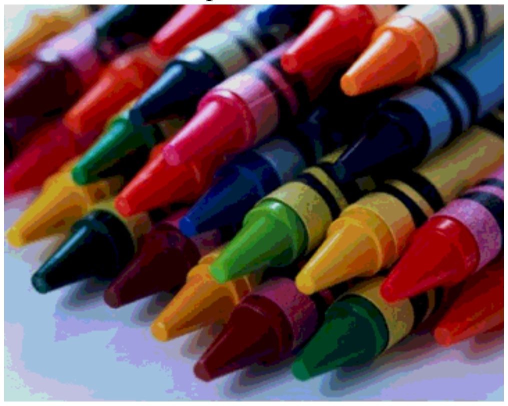
contrast



6. **Posterize**. A typical pixel can have one of 256 value for each color channel. In a posterized image, this number is drastically decreased. Each color channel value should be rounded *down* to the nearest multiple of 32.

original

posterize

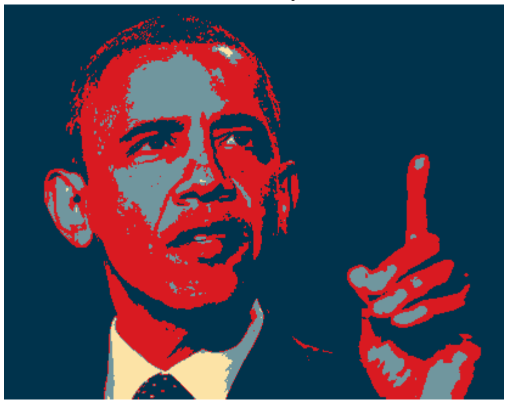


7. **Obamafy**. Every pixel is assigned one of four colors:
   - dark blue with (R,G,B) = (0,51,76)
   - light blue with (R,G,B) = (112,150,158)
   - yellow with (R,G,B) = (252,227,166)
   - red with (R,G,B) = (217,26,33)

   The assignment is based on the pixel's *grayscale value*. Grayscale above 182 is yellow; grayscale between 182 and 121 is blue; grayscale between 121 and 60 is red, and below 60 is dark blue.

original

Obamafy



4 Challenge problem

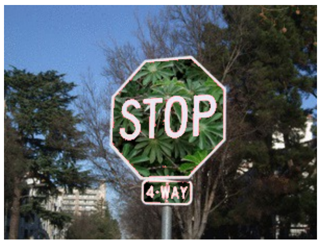
1. **Chroma-key blending** For this problem, you will need to blend *two* images using a "chroma-key." The left and center pictures below show a stop sign and leaves, respectively. What you'll need to do is to create an image similar to the right one, in which the red of the stop sign is replaced by pixels from the image of leaves.

   Chroma key blending works based on pixel color, not position. In other words, *every* "red" pixel in the *entire* image is replaced, regardless of whether that pixel is located within the octagonal stop sign shape. Chroma key blending is also known as "green screening" – e.g., TV weather persons stand in front a green screen, which is then digitally replaced with a weather map. (For more information, see `http://en.wikipedia.org/wiki/Chroma_key`.)

background

foreground

chroma key blended



To perform the blending, you will have to develop a method to detect pixels that are red (or "reddish"). You cannot simply examine the red component of a pixel; you should try to detect situations in which the red component is significantly larger than the other components. *You will need to experiment, trying different notions of "red," until it looks about right.* When you find a red pixel at coordinates x,y in the stop sign image, you can overwrite that pixel with another pixel from the leaves image, retrieved from the same x,y coordinates.

5 An overview of digital images

Digital images are composed of a grid of picture elements called **pixels**. Pixels are essentially tiny dots that are colored in some way. The composition of the grid of pixels, and the various colors of those pixels, is what causes a digital image to have a specific appearance.

Each pixel in the image is located using a simple two-dimensional coordinate system. The `(0,0)` coordinate is located in the upper-left corner of the image (which is somewhat different than you might have expected). If the width of an image is `w` and its height is `h`, the lower-right coordinate is `(w-1,h-1)`. Figure 1 depicts the image coordinate system graphically.

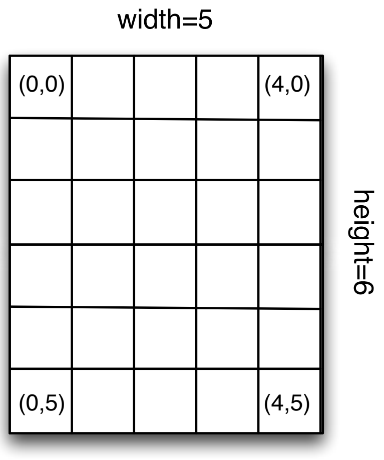


Figure 1: Example showing how the image coordinate system works. The coordinate of each corner pixel is shown along with the width and height of the image.

The color of a pixel is specified using three components: red, green, and blue. The intensity of each color component is what gives a pixel its overall color. Color intensities typically range from 0–255. Examples:

- a pixel colored black would have (R,G,B) intensities of (0,0,0) (no intensity for any color)
- an entirely green pixel would have (R,G,B) = (0,255,0).
- a pixel with (R,G,B) = (255,0,255) has maximum intensity red *and* maximum blue and has a magenta color.
- a white pixel would have (R,G,B) = (255,255,255).
- shades of gray between black and white can be obtained by setting all three colors to the same number for some number between 0 and 255.

With this fairly simple system, one can create a variety of 2^{24} different colors (which is a ridiculously large number of colors!) If you're interested, you can read more about pixels at `http://en.wikipedia.org/wiki/Pixel`.

6 Programming with images

We provide an `image` module that facilitates the process of manipulating images. The `image` module provides functions for creating and saving `Image` objects. The `image` module is distinct from an `Image` object, much like the `turtle` module is distinct from a `Turtle` object.

6.1 Creating, saving, and displaying images

There are three ways to make an Image object.

1. Load an image from a file: `img = image.load_from_file('crayons.png')`. You can open any file in PNG format. If you have a favorite JPG image you want to work with, a google search for "convert jpg to png" will yield links to a number of free online conversion tools. You may also be able to use a program on your computer. (In Apple's Preview program, you can open a JPG file and do "Save As..." to save into a different format.)
2. Make a new image: `img = image.new_image(400, 300)` will make an all black image that is 400 pixels wide by 300 pixels high. You can use the methods on Image objects to change the pixel colors, as explained below.
3. Make a copy of an existing image: suppose `img` is an Image object, then `img_copy = image.copy_image(img)` will create an identical copy of the given Image object.

You can save an Image object named `img` by doing this: `image.save_to_file(img, 'my_img.png')`. The image will be saved to a file named `my_img.png`. The file will be placed in the same folder as the program that you are running.

If you have several images, such as `img`, `img2`, and `img3`, you can display them in a pop-up window by doing this: `image.display_images(img, img2, img3)`. The `display_images` function takes one or more image objects (separated by commas) and displays them. *This statement should be the last statement in your program* because it will halt your program until the user clicks the "Quit" button on the window.

6.2 Working with Image objects

Once you have an Image object, you can call methods on it. First, let's make an an Image object:

```
img = image.load_from_file("crayons.png")
```

Methods `width()` and `height()` return the width and height respectively.

To find out how much red is in an individual pixel, you can use the `get_red` method. For example,

```
r = img.get_red(100, 200)
```

will assign to `r` the amount of red in the pixel at (x, y) = (100, 200). The value will be a number between 0 and 255. Similar methods exist for green and blue.

To change the value of red in an individual pixel, use the `set_red` method. This method takes three arguments. For example,

```
img.set_red(100, 200, 0)
```

will remove all red from the pixel at (x, y) = (100, 200). The third argument to the function must be a number between 0 and 255. Similar methods exist for green and blue.

Sometimes it is convenient to get all three colors at once. For example,

```
r, g, b = img.get_rgb(55, 40)
```

The `get_rgb` method returns *three* numbers – the red, green and blue values – which are in turn assigned to the variables `r`, `g`, `b`. This is special kind of assignment statement called **tuple assignment**. We will learn about this in more detail later in the course. For now, all you need to know is that for this method, you can put three comma-separated variables on the left hand side of the assignment operator. For example,

```
rutabaga, garlic, bok_choy = img.get_rgb(55, 40)
```

will assign the red, green, and blue values to the variables `rutabaga`, `garlic`, `bok_choy` respectively.

Similarly, it is convenient to set all three colors at once. You can use the `set_rgb` method which takes *five* arguments:

```
img.set_rgb(55, 40, 255, 255, 0)
```

the above statement sets the pixel at (x, y) = (55, 40) to be yellow – i.e., the color you get with maximum red and green but no blue.

6.3 Example program #1: red filter

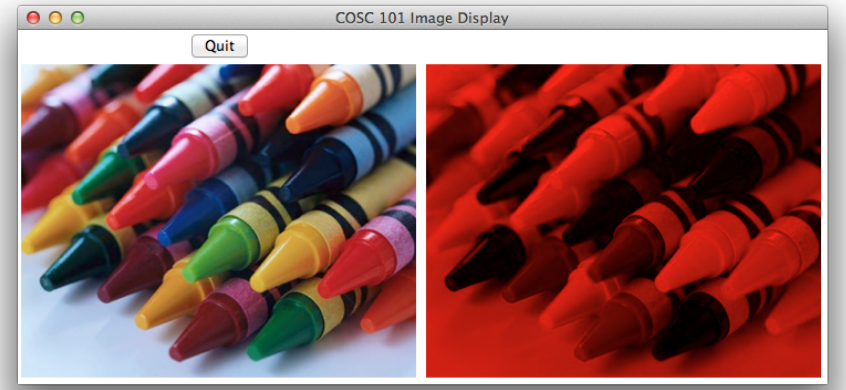
This example program takes the image of the crayons and applies a red filter to it, meaning that all green and blue is removed from the picture and the only light that remains is the light coming from the red channel. As you might expect, crayons that are dark green or dark blue will appear to look almost black in the filtered image.

The program starts by making a copy of the image. This is so that we maintain a copy of the original image so in the end we can display "before" and "after" images side by side. The nested loops iterate over every possible (x, y) coordinate. For each coordinate, we obtain the pixel and remove green and blue by setting them to zero.

```python
import image

img = image.load_from_file('crayons.png')
red_only_img = image.copy_image(img)

w = img.width()
h = img.height()

# loop over every (x,y) pair
```

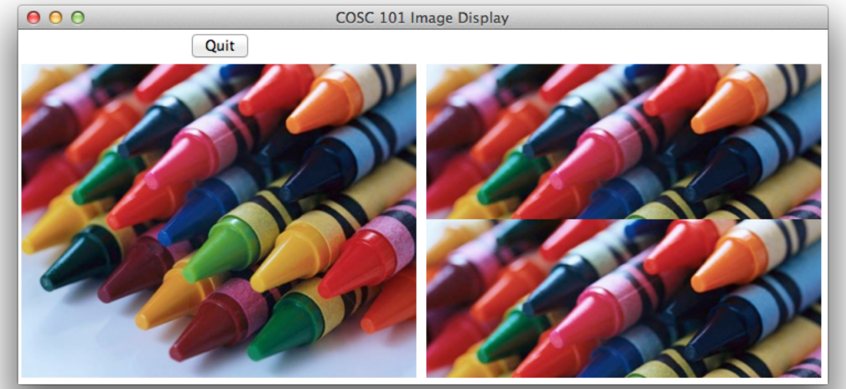
```
for x in range(w):
    for y in range(h):
        # filter out green and blue
        red_only_img.set_green(x, y, 0)
        red_only_img.set_blue(x, y, 0)

# save the new image to a file
image.save_to_file(red_only_img, 'red_crayons.png')

# create window that displays both the original and modified images
image.display_images(img, red_only_img)
```

When run, the program causes this window to be displayed:



6.4 Example program #2: copy top

This example is more complicated than the previous example. In this example, we make a new image of the crayons in which the top half is duplicated and copied over the bottom half. The result looks like this:



Here is the program. Notice the **for** loop for the y coordinate only loops over half of the y values. Also notice that `px_bottom` is a pixel from the bottom half of `img_copy` since it is located at coordinate `(x, y + h/2)`.

```
import image

img = image.load_from_file('crayons.png')
img_copy = image.copy_image(img)

w = img.width()
h = img.height()

# loop over every x value...
for x in range(w):
    # ... but only loop over y values in top half
    for y in range(h/2):
        # get rgb from top half of original image
        r, g, b = img.get_rgb(x, y)

        # copy over rgb from top half to bottom half
        img_copy.set_rgb(x, y + h/2, r, g, b)


# save the new image to a file
image.save_to_file(img_copy, 'copy_top.png')

# create window that displays both the original and modified images
image.display_images(img, img_copy)
```

Acknowledgments: we adapted these assignments from previous assignments by Tom Wexler, Nick Parlante, Mark Guzdial.