Lab 10 – Transforming Your Image

The Way of the Program - Fall 2023

Download **lab10_files.zip** from the class web page under Labs and unzip it on your Desktop. The folder contains a collection of GIF files that you can use for image transformations, as well as the `pix` graphics module. The starting code for the lab is in **startcode.py**, which contains the functions we wrote in class yesterday.

1. Using these functions as a guide, write a new function **darken(*img*)** that takes an image and darkens each pixel by multiplying each RGB value by 0.5. Also write a function called **lighten(*img*)** that lightens each pixel by doubling the RGB values. You will need to use the `clip` function to make sure that the doubled values stay within the range 0-255.

2. Next, write a general-purpose function called **brightness(*img*, *factor*)** that changes the brightness of an image by multiplying each RGB value by the *factor* parameter. For example, you should be able to achieve the same effect as **darken** and **lighten** by calling brightness(*img*, 0.5) and brightness(*img*, 2), respectively.

3. Write a function **addnoise(*img*)** that takes an image and adds some random noise to each color channel—that is, to each RGB value. There are many ways you could do this. For example, you could generate a single random number in the range -100 to +100, and add it to the three RGB values (make sure to use `clip` to keep the values in the range 0-255), or you could generate three different random numbers and add each one separately to R, G, and B. Try it both ways to see which way looks "noisier". What if you use a different random number range, or add noise to just one color channel instead of all three?

4. In class we wrote the **reflectLeft** function that "reflects" an image horizontally by copying the pixels in the left half of each horizontal row to their symmetric positions in the right half of the row. Write a new function called **reflectRight(*img*)** that copies the pixels in the opposite direction, from the right half of each row to the left half. For this version, have `x` start in the *middle* of the row and go to the right, while `oppositeX` starts in the middle and goes left. Try out your program on **santa.gif**. Do you like this version of Santa any better?

5. The **red(*img*)** function currently sets all of the pixels of an image to red. Modify it so that it sets only the *top half* of the image to red. Hint: you will need to modify one of the nested for-loops appropriately, using integer division (`//`) to avoid introducing floating-point values into the range.

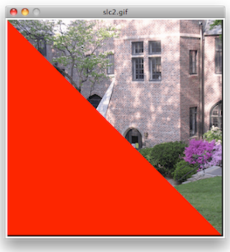
6. Modify **red** so that it sets only the *left half* of the image to red.

7. Modify **red** so that it sets only the *bottom half* of the image to red. Hint: you'll need to specify a different starting value in one of the for-loops, in addition to an ending value.

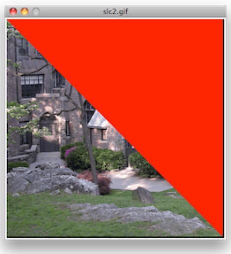
8. Modify **red** so that it sets only the *right half* of the image to red.

9. Now change **red** back to the original version, so that it sets the entire image to red, before going on to the exercises below. Do these exercises in the order shown. Also make sure to use the square test image **slc2.gif**.

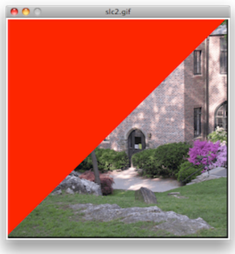
   a) Modify **red** so that it sets the "southwest triangular" region to red, as shown in figure (a) below.
   b) Modify **red** so that it sets the "northeast triangular" region to red, as shown in (b).
   c) Modify **red** so that it sets the "northwest triangular" region to red, as shown in (c).
   d) Modify **red** so that it sets the "southeast triangular" region to red, as shown in (d).



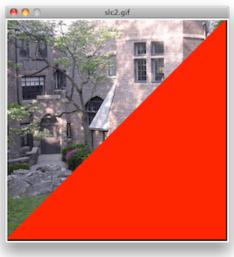
(a)



(b)

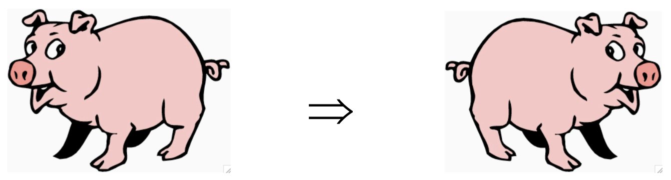


(c)



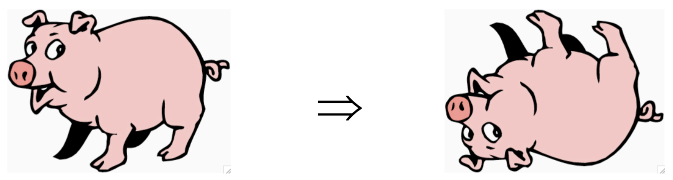
(d)

10. Write a function called **hflip(*img*)** that takes an image and flips it horizontally. For example:



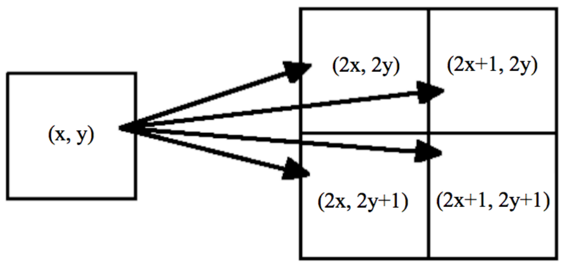
One way to do this is to use a nested loop structure that loops through each row of the image, starting with the top row and working down to the bottom row. For each row, the outermost two pixels are swapped, then the next two inner pixels, then the next two, and so on until we come to the middle of the row. Then the outer loop continues with the next row. To aid debugging, it might help temporarily to include a call to `img.update()` immediately after changing each pixel, so that you can see the order that the pixels get updated. After you are sure your loop works correctly, you can move the `update` call to the end of your function, if you prefer, so that all the pixels get updated at once, right before the function terminates.

11. Write a function called **vflip(*img*)** that takes an image and flips it vertically. For example:



This function will be similar in structure to `hflip`, except that this time we will process the image column by column. For each column, we swap the top and bottom pixels in the column, then the next two inner pixels, and so on, until we reach the middle of the column. Then we continue with the next column.

12. Next, write a function called **enlarge(*img*)** that creates a new enlarged version of the given image. To do this, first construct a new blank image with a width and height twice as big as the original. You can find out the dimensions of an image by calling `img.getWidth()` and `img.getHeight()`. Then go through the original image pixel by pixel: for each pixel *p* at location (*x*, *y*) in the original, you will need to color *four pixels* in the new image the same color as *p*. See the diagram below. After all of the pixels in the new image have been initialized, the new image should be returned.



13. The method `img.getMouse()` returns a Point corresponding to the location clicked by the mouse in a Picture window. Using `getMouse`, write a function called **crop(*img*)** that waits for the user to click on two points, then creates and returns a new image consisting of the pixels within the rectangular region delimited by the mouse clicks. For example, typing `img2 = crop(img)` at the Python prompt and then clicking on the upper left and lower right corners of a rectangular region within `img` should create a new image `img2` containing just that region.

14. Write a function **soften(*img*)** that creates and returns a new image of the same size and sets each pixel in the new image to be the *average color* of the surrounding pixels in the original image. More precisely, the new pixel's R value should be the average of the R values of the pixels to the north, south, east, and west in the original image; the new G value should be the average of the original neighboring G values, and so on.