CS 1120: Media Computation Spring 2017
Lab 2: Manipulating photos with loops

Getting Started

In my code below I will be using an image file, beach.jpg, available at
http://sergey.cs.uni.edu/courses/cs1120/mediasources/beach.jpg
The first time through this lab I ask you to use that image as well so that our answers are consistent. But I will later ask you to use any second photo of your choice. You may use any photo of your choice. My only suggestion is that you use one whose dimensions aren't much larger than 500 pixels in either direction.

Activity A: Getting started with functions that manipulate entire photos

Launch JES. Now, type the following program into the Program Area of JES and save it as "lab2.py". Make sure that you pay attention to the spelling and the spacing of the different features.

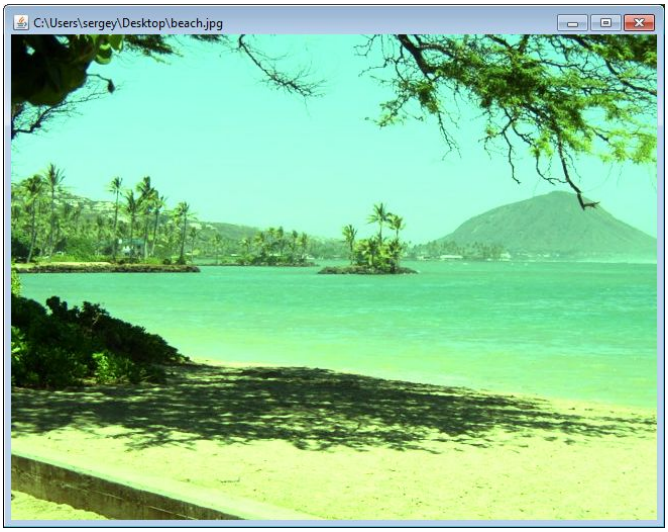
```
1  def increaseGreen(picture):
2    for pixel in getPixels(picture):
3      greenValue = getGreen(pixel)
4      setGreen(pixel, greenValue * 1.5)
```

When you have it done re-save and load this file.
Fix any syntax errors you may have made.
When everything loads properly type the following commands:

```
======= Loading Program =======
>>> setMediaPath()
'C:\\Users\\sergey\\Desktop\\'
>>> original = makePicture("beach.jpg")
>>> changed = makePicture("beach.jpg")
>>> increaseGreen(changed)
>>> show(original)
>>> show(changed)
```

When you invoke setMediaPath() you will need to navigate to wherever your images are stored.

At this point two pictures should be on the screen - although changed may have popped up right over the top of the original photo. Move them side by side so you can compare them:
Can you see the difference? Mine looked like this::



Suppose that we wanted to confirm that they really ARE different however. Well, then we might type:

```
>>> explore(original)
>>> explore(changed)
```

This pops up two windows that look similar to the ones above, but that allow you to identify the colors at specific pixels. Let's consider one spot out on the water. Using the boxes at the top of each explore window enter the pixel (225,250).

[Q1] What are the RGB color values of the pixel from the original image?

[Q2] What are the RGB color values of the pixel from the changed image?

[Q3] Compare the red values between these two pixels. How do they compare?

[Q4] Compare the green values between these two pixels. How do they compare?

[Q5] Compare the blue values between these two pixels. How do they compare?

If you did everything correctly you should notice that red and blue stayed the same but that green changed. In fact, the green value of the changed image is exactly 1.5 times that of the green value in the original image. You might expect that is true for all pixels, but in fact, it is not.

Repeat this process with the pixel (100,400).

[Q6] What are the RGB color values of the pixel from the original image?

[Q7] What are the RGB color values of the pixel from the changed image?

[Q8] Notice that the green value in the changed pixel is NOT 1.5 times the green value of the original pixel. Why do you think that is (HINT. What number is 1.5 times the green value of the original pixel? Why can't the computer set to that value?)

There are times that you will want to do what we just did using the explorer windows but there are other times where it will be more helpful to do this with code/commands instead. Type in the following:

```
>>> originalPixel = getPixel(original, 100, 100)
>>> print originalPixel
Pixel red=160 green=204 blue=231
>>> changedPixel = getPixel(changed, 100, 100)
>>> print changedPixel
Pixel red=160 green=255 blue=231
```

Activity B: Understanding how this program works

At first it might look like this program just "magically" changes all of the pixels to have a higher value for the green channel. But this actually isn't correct. It actually is doing this one pixel at a time.

Look at the line of code that says: `for pixel in getPixels(picture):`

Let's start with the last part of that command `getPixels(picture):`

This command asks the computer to generate a giant list of every single pixel in the picture. It's big. As the book says, you wouldn't want to try and look at that or print it out. Think about it. The beach picture is 640x480 which means that it contains 307,200 different pixels. It would take a lot of time to print details about every one of those pixels. But you can trust us when we tell you that this command makes a BIG list of every single pixel.

The rest of the command - `for pixel in` - basically asks the computer to consider each of those pixels one at a time, storing the one that you are looking at in a variable named pixel. This means that one at a time you are getting a pixel, storing it in a variable called pixel, getting the green value from that pixel, and changing the green value of that pixel. After the `setGreen()` command is done the computer comes back to the top of the for loop and sets the variable named pixel to point at the NEXT pixel in the list. In this manner, the computer looks at each pixel one at a time and changes it's green value.

While I don't recommend doing this very often, I do like to have you do the following once in a while so that you can understand what is going on. Modify your previous program so that it says:

```
def increaseGreen(picture):
  for pixel in getPixels(picture):
    greenValue = getGreen(pixel)
    setGreen(pixel, greenValue * 1.5)
    repaint(picture)
```

Notice what this change does. It asks the computer to repaint the picture after EVERY SINGLE change that is made. In other words, we are going to repaint the beach picture 307,200 times.

You can see why I wouldn't want you to do this very often, but by doing it once in a while you can see what is happening.

Run this code by typing:

```
>>> changed = makePicture("beach.jpg")
>>> show(changed)
>>> increaseGreen(changed)
```

It will probably take a few seconds before you see anything happening. But after a little bit you should notice that the photo is slowly changing one row at a time. (When you get bored of watching this change you can press the "stop" button which is on the right side of the screen on the bar between the Program Area and the Command Area).

Activity C: Exploring some mystery functions

Begin by simply READING the program shown below:

```
def mysteryMethod1(picture):
  for pixel in getPixels(picture):
    redValue = getRed(pixel)
    setRed(pixel, redValue * 0.5)
```

[Q9] What do you PREDICT this code will do?

After you make your prediction, enter the program into lab4.py, save, and load. Fix any typos you may have made. Run the program and see what actually happened by doing something like:

```
>>> original = makePicture("beach.jpg")
>>> changed = makePicture("beach.jpg")
>>> mysteryMethod1(changed)
>>> explore(original)
>>> explore(changed)
```

[Q10] What did the code ACTUALLY do? How did you verify this?

[Q11] mysteryMethod1() is a bad name. It doesn't tell us anything about what this does. What would be a better name?

Now let's repeat this process with some other mystery methods…

Begin by simply READING the program shown below:

```
def mysteryMethod2(picture):
  for pixel in getPixels(picture):
    setRed(pixel, getRed(pixel) * 0.5)
```

[Q12] What do you PREDICT this code will do?

After you make your prediction, enter the program into lab4.py, save, load, and test.

[Q13] What did the code ACTUALLY do? How did you verify this?

[Q14] mysteryMethod2() is a bad name. It doesn't tell us anything about what this does. What would be a better name?

Begin by simply READING the program shown below:

```
def mysteryMethod3(picture):
  for pixel in getPixels(picture):
    setBlue(pixel, 0)
```

[Q15] What do you PREDICT this code will do?

After you make your prediction, enter the program into lab4.py, save, load, and test.

[Q16] What did the code ACTUALLY do? How did you verify this?

[Q17] mysteryMethod3() is a bad name. It doesn't tell us anything about what this does. What would be a better name?

Begin by simply READING the program shown below:

```
def mysteryMethod4(picture):
  for pixel in getPixels(picture):
    r = getRed(pixel) * 1.2
    g = getGreen(pixel) * 1.2
    b = getBlue(pixel) * 1.2
    newColor = makeColor(r, g, b)
    setColor(pixel, newColor)
```

[Q18] What do you PREDICT this code will do?

After you make your prediction, enter the program into lab4.py, save, load, and test.

[Q19] What did the code ACTUALLY do? How did you verify this?

[Q20] mysteryMethod4() is a bad name. It doesn't tell us anything about what this does. What would be a better name?

Begin by simply READING the program shown below:

```
def mysteryMethod5(picture):
  for pixel in getPixels(picture):
    r = getRed(pixel) + 20
    g = getGreen(pixel) + 20
    b = getBlue(pixel) + 20
    newColor = makeColor(r, g, b)
    setColor(pixel, newColor)
```

[Q21] What do you PREDICT this code will do?

After you make your prediction, enter the program into lab4.py, save, load, and test.

[Q22] What did the code ACTUALLY do? How did you verify this?

[Q23] mysteryMethod5() is a bad name. It doesn't tell us anything about what this does. What would be a better name?

Begin by simply READING the program shown below:

```
def mysteryMethod6(picture):
  for pixel in getPixels(picture):
    r = getRed(pixel)
    g = getGreen(pixel)
    b = getBlue(pixel)
    newColor = makeColor(b, r, g)
    setColor(pixel, newColor)
```

[Q24] What do you PREDICT this code will do?

After you make your prediction, enter the program into lab4.py, save, load, and test.

[Q25] What did the code ACTUALLY do? How did you verify this?

[Q26] mysteryMethod6() is a bad name. It doesn't tell us anything about what this does. What would be a better name?