The goal of this seventh Python programming assignment is, one last time, to have you extend your previous PhotoSlop Elephants project to add new functionality. You are to perform this assignment in the PC lab at your scheduled lab time.

**The Goal**

Using your *existing program*, modify and extend your program to include the following functions, with the command codes as indicated (update the message prompt appropriately in each case). A sample transformation is shown in each case.

<table>
<thead>
<tr>
<th>Command(s)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>Rotate the image counterclockwise by 90°.</td>
</tr>
<tr>
<td>)</td>
<td>Rotate the image clockwise by 90°.</td>
</tr>
<tr>
<td>d</td>
<td>Dither the existing image to black and white.</td>
</tr>
<tr>
<td>D</td>
<td>Dither the existing image to RGB (8 color).</td>
</tr>
</tbody>
</table>
Rotating Images

You will need to create and return a new canvas as the result of each rotation function. That is, the basic framework of one function should be:

```python
def RotateClockwise (Canvas):
    NewCanvas = makeEmptyPicture(___,___)
    return NewCanvas
```

and the call to this function from the command processor in your main program will be:

```python
Canvas = RotateClockwise (Canvas)
```

The other function will be similar. The big issue you must address is that an arbitrary image is not likely to be square, meaning that its width and height will be different. That affects the size of the new canvas that you create, as well as how the pixels from the original canvas are copied into the new canvas. For example, an original image that is 300×200 pixels will return an image that is 200×300. The pixels from a row of the source canvas need to be copied into a column of the new canvas, so you must correctly transpose rows and columns into columns and rows, without running off the edge of either canvas.

There is not a graceful way to show the progress of the rotations, as we will be replacing one canvas with another. Do not use the repaint function in either rotation function; simply have your program “pop” the transformed image onto the screen when it is done.

Dithering Images

Posterization to a reduced set of colors is something you did in part 1 of this assignment. In those functions, you mapped the color of every pixel onto a reduced set of colors, either monochrome (black and white) or eight-color RGB. In dithering, the initial approach is the same (replace each pixel with a new color), but with the additional step of distributing the differences between the old pixel value and the new pixel value to pixels not yet processed. While the number of colors in the image has been massively reduced, the average color or shade of gray in the neighborhood of each pixel is close to its original value. That means that if you squint at a dithered image the “fuzzing out” you see will re-blend the colors back to close to what they were.

The particular type of dithering we will use (and there are many methods) is called Floyd-Steinberg dithering (http://en.wikipedia.org/wiki/Floyd_Steinberg). As with most of the in-place image processing functions that you have written, you will be scanning the image row by row, pixel by pixel, from the top left to the bottom right. As you process each pixel, you must compute the differences in R, in G, and in B between the old pixel value and the new pixel value. Those differences will be distributed (added to) the R, G, and B of the neighboring pixels not yet processed: the E (east) neighbor, the SW neighbor, the S neighbor, and the SE neighbor.
This is shown in the following image:

Each of those four neighbors will get just a fraction of the error: $\frac{7}{16}$ of the error goes to the E neighbor, $\frac{3}{16}$ to the SW, $\frac{5}{16}$ to the S, and $\frac{1}{16}$ to the SE. That means you must do this three times for each pixel, once for red, once for green, and once for blue.

For example, let's say that the current pixel is (R=192, G=37, B=247), and we dither it to RGB as (R=255, G=0, B=255). The error in the red is 192-255=-63, the error in the green is 37-0=37, and the error in blue is 247-255=-8. Therefore, $\frac{7}{16}$ of -63 is added to the E neighbor’s red, $\frac{7}{16}$ of 37 is added to the E neighbor’s green, and $\frac{7}{16}$ of -8 is added to the E neighbor’s blue, $\frac{7}{16}$ of -63 is added to the SW neighbor’s red, and so on.

You must be careful to not reference pixels that are off the edges of the canvas: you cannot add error to a non-existent pixel, so you must check for and ignore those cases.

As these are in-place image transformations, you can (and must) show the progress of the function as it operates.

**Extra Credit (+10 Points Possible)**

You can up to double your score on this lab by doing extra credit. Further extend your program to include one or both of the following defined functions (+5 points each), with the command codes as indicated (update the message prompt appropriately in each case).

<table>
<thead>
<tr>
<th>Command(s)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td><strong>Scale_Up</strong> Scale the image up by a factor of 2. That is, create a new canvas with dimensions <code>getWidth(Canvas)*2</code> and <code>getHeight(Canvas)*2</code>, and then copy the color of each pixel from the original canvas to the appropriate 2×2 block of pixels in the new canvas.</td>
</tr>
<tr>
<td>/</td>
<td><strong>Scale_Down</strong> Scale the image down by a factor of 2. That is, create a new canvas <code>getWidth(Canvas)/2</code> by <code>getHeight(Canvas)/2</code>, and then copy the average color of each 2×2 block of pixels from the original canvas to the appropriate destination pixel.</td>
</tr>
</tbody>
</table>

Test your code.
**Finishing Up**

That is enough information for you to figure out how to fill out the functions and run the program. Test each function. Demonstrate your program to the TA and show that each function works correctly.

Download from the class site the image of the small snake. Run the program and load in that image, and then follow these steps:

- **D** Rotate the image counterclockwise.
- **d** Dither the image to B&W.

Quit the program, but leave the final image on-screen. Run the program a second time, load in the snake image once more, and then follow these steps:

- **D** Rotate the image clockwise.
- **d** Dither the image to RGB.

Quit the program.

When you are finished and everything runs correctly, print out (File-Print) the source code to turn in. Also, capture and print a screen shot that shows your result from running the program – make certain that before you capture the screen the JES window shows as much of your code as possible (including the part with your name) AND the result of both executions (the two processed images of the snake), along with the folder containing both the original image AND the transformed image. It may be difficult to arrange all this neatly on-screen, but do the best you can.

If you do any of the extra credit, run the program once for each of the extra credit pieces that you implemented. If you implement the **Scale_Up** function, run the program, load in the snake image, scale it UP, and capture a new screen shot. If you implement the **Scale_Down** function, run the program, load in the snake image, scale it DOWN, and capture a new screen shot. Turn the screen shot(s) in with the rest of your assignment.

(In Windows, hit the Prt Sce or Prnt Scrn or PrtSc button to copy the image to the clipboard, paste the image into Windows Paint (or your favorite image processing program), and print it from there; it may come out on multiple pages. On the Mac, hitting the key combination Cmd-Shift-3 will save the screen as a .png file on the desktop; double-click the .png file to bring it into the Preview program, and then print it from there.)

Turn in to the TA the printout from JES and the screen shot(s), **stapled** together in that order (the source code version printed from JES on top, the main screen shot image next, and any extra credit screen shots on the bottom).