Introduction

The ninth assignment for 600.112: Introduction to Programming for Scientists and Engineers is all about image processing using the PILLOW library.

There are three things to do: First you’ll write a program that can be used to create ANDY WARHOL-style “pop art” from a given image. Second you’ll write a program that can “unmask” images hidden inside an otherwise random-looking picture. Third you’ll write a program to produce image mosaics: A lot of small images put together to resemble another image when looked at from a distance.

There are detailed submission instructions on Piazza which you should follow to the letter! You can lose points if you create more work than necessary for the graders by not following the instructions.

1 Automated Pop Art (20%)

ANDY WARHOL was an American artist and a leading figure in the visual art movement known as pop art. Among many other things, Warhol is famous for images like the one in Figure 1 which can be seen as a commentary on the “celebrity culture” of “mass-produced personalities” created by the media especially since the 1950s. For this problem, you are to develop a program that “approximates” this style of art in a simple way. Given an original image, your program should do two things:

- produce an image of approximately the same size, but containing the original replicated nine times in a pattern similar to Figure 1
- change the colors of each sub-image by modifying the color bands of the original image in a suitably strange way
Figure 2 shows an example of this process: your program is to turn the original image of JACK NICHOLSON on the left into the “pop art” image on the right. Please call your program warhol.py, nothing else. The name of the original image will be supplied as a command-line argument by the user, so you’ll have to test your program in the UNIX shell. Your program will save the resulting pop art image into a file called warhol.jpg, nothing else.

A rough outline of the program we have to write would be as follows:

- load the original image named by the user
- shrink the image to one-third of its original size
- produce nine copies of the smaller image, each with a different color scheme
- assemble the nine smaller images back into one larger three-by-three image
- save the resulting pop art image

In order to produce the different color schemes by exchanging the red, green, and blue color bands, the original image must of course have those color bands. This leads us to a first outline for the program as follows:

```python
def main():
    if len(sys.argv) != 2:
        print("error: need exactly one argument")
        exit()

    original = PIL.Image.open(sys.argv[1])

    if original.mode != "RGB":
        print("error: need image with mode RGB to remix")
        exit()

    width, height = original.size
    reduced = original.resize((width // 3, height // 3), PIL.ANTIALIAS)

    images = remix_all(reduced)
    warhol = assemble(images)

    warhol.save("warhol.jpg")
    warhol.show()

if __name__ == "__main__":
    main()
```

All of the PILLOW functions (and methods) used here were discussed in lecture. The ANTIALIAS filter for the resize method improves the quality of the reduced image, for details see the PILLOW documentation.

The idea for the remix_all function is to return a list of nine images, each with its color bands modified as required to produce the pop art look we’re going for. The idea for the assemble function is to take a list of nine images (all of identical size) and to produce the “composite” image that arranges them in a three-by-three layout.

Let’s tackle the second function first. We need to create a new image of three times the size of the
Assemble given images as a 3x3 grid.

Make sure you understand how the coordinates for the paste operation are computed, it’s not totally obvious at first that this is the correct way.

Now let’s tackle the remix_all function that produces the images with different color schemes by exchanging the color bands of the original image in some way. Obviously we first need to decide how we want to actually transform the color bands. Remember that the split method for an image in PILLOW returns three new images, one for each of the color bands red, green, and blue. The merge function does the reverse: it takes three color bands and puts them together into a complete image. We saw in lecture that by splitting an image and then merging the bands together in a different order again, we can achieve interesting color effects.

For our pop art we need nine such rearrangements, and obviously it would be a bad idea to write a complicated if instruction with nine different cases. We should do what we’ve done before for the Knight’s Tour program: Encode these rearrangements as data and the write some code to follow those instructions. Here’s one way to express the knowledge of how we want to rearrange the color bands for each of the nine small images:

```
MIXES = (  
    "BGR", "GRB", "BRR",  
    "BRG", "RGB", "RBR",  
    "GBR", "GRR", "GBB")
```

We read this as follows: “To rearrange the color bands of the top-left image, use the original’s blue band for red in the pop art, use the original’s green band for blue in the pop art, and use the original’s red band for green in the pop art.” The other eight versions can be read the same way, and since there are only six permutations of three things (without repetitions) we’ll have to have at least a few cases in which we use a color band twice.

With the MIXES data structure in place, we can now write the remix_all function:

```
def remix_all(image):
    """
    Return nine remixed images.
    """
    bands = image.split()
    result = []

    for mix in MIXES:
        new = remix_one(mix, bands)
        result.append(new)

    return result
```

As usual we concentrate on one task in each function: Here we build the list of pop art images based on the required mix for each, but we put the knowledge of how exactly we perform the remixing of one image into a separate function remix_one. Note, however, that we split the image into its constituent bands once in this function. We could have just as well performed the split in remix_one (and I actually did it that way in class because I was out of

1. Note that we’ve layed out the data in a way that reminds us of the pop art image we want to produce; if there is a simple structure to a problem that we can reproduce in code, it’s usually a good idea to do so.
time), but then we would have done the exact same operation nine times instead of once, an obvious inefficiency.

Here, finally, is the `remix_one` function that mucks around with the color bands:

```python
def remix_one(mix, bands):
    # Remix color bands according to mix.
    red, green, blue = bands
    mixed = []
    for band in mix:
        if band == "R":
            mixed.append(red)
        elif band == "G":
            mixed.append(green)
        elif band == "B":
            mixed.append(blue)
        else:
            print("error: invalid band ← {} for remixing".format(band))
            exit()
    result = PIL.merge("RGB", mixed)
    return result
```

Note the error checking: If someone put the wrong kind of data into the `MIXES` tuple, we want to notice that mistake, not just ignore it silently. There’s a slightly nicer way of writing this function without the big if cascade by using a dictionary. If you feel strongly about improving the code, feel free to replace the if but remember that you still want to be able to check for the same kind of error...

By the way, funny things tend to happen when you run your program on its own output, see Figure 3. Here’s to hoping that you’ll have some fun with this program on Facebook!

2 Hidden Images (30%)

According to Wikipedia, *steganography* is the “art or practice of concealing a message, image, or file within another message, image, or file” and in this problem you’ll deal with a very simple technique to achieve just that. Figure 4 shows an image of blue and green dots (right) as well as the image hiding within it (left). How was this done?

As you know, most color images are composed of three different color bands: red, green, and blue. Each band by itself is essentially a grey scale image because it only specifies the intensity (from 0 to 255) of one band of color. So in order to “hide” an image inside another one, a simple approach is as follows:

- convert the image we want to hide to grey scale
- create a new image with randomly filled green and blue bands, but set the red band equal to the grey scale image we want to hide
- save the resulting color image

Figure 3 Facing 81 (left) or 729 (right) Jacks.

Figure 4 Hiding Mini-Me (left) in chaos (right).
Actually that’s not quite enough, especially if the image we want to hide is bright: Although it would be difficult to make out details, people could still tell that there is a pattern beyond just random noise. So we also reduce the brightness of the image we want to hide significantly, for example to a point where only the lowest 15 levels of grey are used. Then the variations in the resulting color image are too small (for most people anyway) to detect the pattern.

Your task for this problem is to write a program that can automatically “break” this simple approach to steganography. Given any image produced using the above technique, your program will identify the channel the “hidden” image is in and display it. As mentioned in lecture, it may be a good idea to read through the documentation for the ImageStats, ImageEnhance, and ImageOps modules for this problem...

Please call your program unmask.py, nothing else. The name of the original image will be supplied as a command-line argument by the user, just like in Problem 1. Your program will save the hidden image it recovered into a file called hidden.jpg, nothing else; it will also display the image.

3 Image Mosaics (50%)

You have probably all seen mosaics of images: A big image made up of many small images. If looked at from a distance you see one thing, if looked at closely you see many different things.

Figure 5 shows the now familiar face of Jack Nicholson as two different mosaics. The one on the left is made up of monochromatic “tiles” chosen from a large variety of possible colors, the one on the right is made up of further images of Jack but a much smaller variety of them. Obviously a larger variety of tiles (a “bigger library”) results in a smoother mosaic, but actual images of Jack are more interesting to look at closely.

Mosaics are constructed from a target image and a library of tiles. The simplest approach divides the target image into small square chunks, chunks of 16-by-16 pixels for example. For each such chunk the average color is computed; that’s the target color for the chunk in question. For each image in the library we also compute its average color. Now, for each chunk of the target image, we find the library image that matches the target color most closely. We then replace that chunk of the target image with a suitably resized version of the library image. Once this has been done for all chunks of the target image, we have our mosaic. Easy.

Your task for this problem is to write a program that given a target image and a library of tiles will produce a decent mosaic. We’ll provide some helper code for you on Piazza that allows you to read an entire directory full of images (the library) and also provides a function to compute the average color of (part of) an image. You are responsible for developing the remaining code: dividing the target image into chunks, computing the average colors of all chunks and all library images, performing the actual matching of chunks to tiles, and producing the final image mosaic.

2. If you look at colors as vectors, any distance metric on vectors will be suitable: We’ll find the library image with the smallest distance. If you need a reminder about distance metrics, check Assignment 5.
Please call your program `mosaic.py`, nothing else. The name of the target image as well as the name of the directory containing the library of tiles will be supplied as a command-line arguments by the user. Your program will save the resulting image mosaic into a file `mosaic.jpg`, nothing else; it will also display the mosaic.