Problem Set 5: Using Libraries

Pset Buddy

You do not have a buddy assigned for this pset.

Introduction

The problem set will introduce you to the topic of library use, that is, using existing libraries in order to accomplish a goal. This problem set involves writing very little code; the learning process is finding useful functions that do the work for you in the existing libraries. You are expected to make use of web search to locate the necessary functions.

Collaboration:

You may work with other students. However, each student should write up and hand in his or her assignment separately. Be sure to indicate with whom you have worked in the comments of your submission.

Note on Grading:

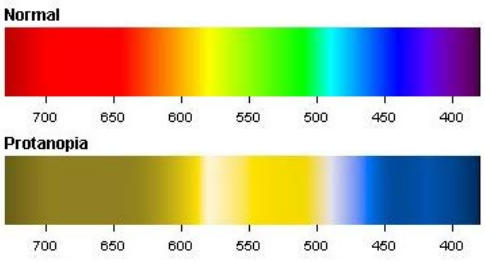
This assignment will have a higher weight to the manual grading as the autograder is not as extensive. 50% of the grade will be from the autograder and 50% will be from manually grading the produced images.

The tester **test_ps5_student.py** will help you determine if you are revealing the hidden images correctly.

1) Colorblindness Filters

The human eye contains two types of photoreceptive (light sensitive) cells: rods and cones. Rods are responsible for vision in low light environments, such as at night, and cones are responsible for color vision. There are three types of cones, each responsible for a particular color, that is, red, green and blue. The red, green and blue cones all work together allowing you to see the whole spectrum of colors. For example, when the red and blue cones are stimulated in a certain way, you will see the color purple.

The condition known as colorblindness typically manifests as a deficiency in one of these types of cones, for example, protanopia is a lack of sensitivity to red light, so the following is an example of the difference in viewing an image.



The objective of this part is to create filters to simulate these differences. To do this, we will make use of Python's Image Library or PIL for short. From PIL we will import `Image`, which is a sublibrary, or a set of functions and methods, related to image processing. With `Image`, we can convert images to a list of pixels, and manipulate the RGB (red, green, blue) values of these pixels.

We can replicate the effects of colorblindness with a matrix multiplication between a colorblindness transformation matrix and the RGB values of a particular pixel, which are represented as a vector with 3 entries.

You do not need to know anything about matrix multiplication; however, if you wish to pursue more information on it, see here.

Here is a general breakdown of the image transformation process:

1. Open the image in Python
2. Retrieve a list (of tuples or ints) of the pixel information
3. Iterate through pixels and multiply them by the transformation matrix using `matrix_multiply`
4. Save transformed pixels as an image

1.1) Implementing Helper Functions

1.1.1) img_to_pix(filename)

The first helper function will be to convert the image to a list of pixels. The input is a string representing an image file, like `'example.jpg'`, and the output is a list corresponding to the pixels in that image, e.g.: `[(0,0,0),(255,255,255),(38,29,58)...]` for RGB, `[60, 66, 72...]` for BW. The list will consist of 3-element tuples that correspond to the RGB values of that pixel for a RGB image, or an integer corresponding to the brightness of that pixel for a BW image.

You'll want to open the image and get the data of that image. (Check the Image module documentation for how to do these things.) Note: Don't worry about determining if an image is RGB or BW. The PIL library functions you will use return the correct pixel values for either image mode.

This should be **very short** if you can find the appropriate functions.

1.1.2) pix_to_img(pixels_list, size, mode)

The next helper function is to convert a list of pixels to an image. The input will be pixels_list, in the same format as the output of the previous function, a size parameter which is a two-element tuple that describes the dimensions of the output image, and a mode parameter which determines whether the input pixels represent a black and white or a color image. Assume that size is a valid input such that `size[0] * size[1] == len(pixels)`.

In other words, you want to **copy pixel values from a sequence object into the image**, so you should check the Image module documentation on how to do this. See the docstring for details on how mode should be represented.

This should be **very short** if you can find the appropriate functions.

1.1.3) filter(pixels_list, color)

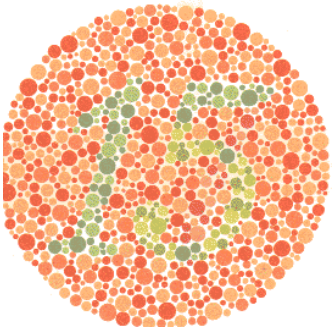
This function takes in a list of pixels in RGB form, such as `[(0,0,0),(255,255,255),(38,29,58)...]`, as well as a color: 'red', 'blue', 'green', or 'none'. The purpose of this function is to apply a transformation to the pixels in the input list that simulate impairment in the cones of the input color. Return the list of transformed pixels.

In order to do the transformation, multiply each pixel by the appropriate matrix. We have provided a `make_matrix` function that will supply a matrix representing the appropriate transformation depending on the input string. For example, `make_matrix('red')` will return the matrix representing a red deficiency in one's vision.

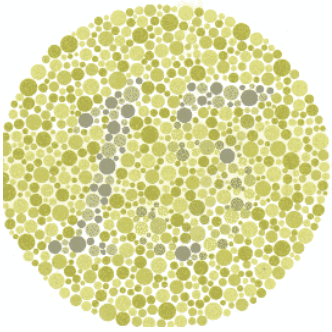
Remember that you must **iterate through the list of pixels and multiply each one by the appropriate matrix**. Use the provided `matrix_multiply(matrix1, matrix2)` method to do this. Please keep in mind that we want the **transformation matrix to be the first argument and the RGB pixel vector to be the second**, or else the resulting image will be incorrect. Take note that `matrix_multiply` returns a **list of floats** and that the pixels **must be tuples of ints**.

This should be **fairly short** if you can find the appropriate functions.

If all is done correctly, then running the program with the test image `('image_15.png')` and a red deficiency should result in the following:



The original image



The transformed image

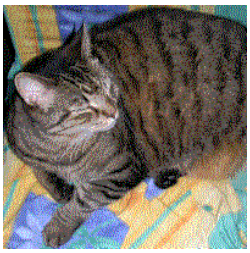
Images like these are known as Ishihara tests.

To receive credit, submit the transformed image at the end of the PSet.

2) Hidden Messages in Images

In an image, it is possible to hide a second (secret) image that can only be recovered by certain mathematical operations. This is known as steganography. In this section, you will go through some examples and write your own steganography module to hide images.

The original image



The secret image

An example of steganography is provided above. The image on the left contains the image on the right. When you are done with this problem set, you will understand how to extract the secret image from the original.

In a black and white image, each pixel is represented by a numerical value that indicates its brightness. The minimum value for each pixel is 0 (which corresponds to black), while the maximum value (which corresponds to white) depends on the number of bits used for each pixel. In our examples, we will use 8-bit images. This means the pixels can take values between 0 and 255, inclusive (can you see why?).

For this part, assume you are dealing with images that have 8-bit color depth.

2.1) Working with Binary Numbers

This part of the pset relies on reading and manipulating binary numbers.

As an example, let's take a look at the binary representation of the decimal number 13, which is 1101. Starting from the rightmost digit in a binary number and moving left, each index location represents an increasing power of 2:

$$1 \quad 1 \quad 0 \quad 1$$

$$2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

When we sum up these powers of 2 (i.e. $2^3 + 2^2 + 2^0$), we get 13.

For more examples and explanations of binary numbers, see lecture slides from Lecture 3.

2.1.1) Least Significant Bits

The most common technique for embedding secret images is to use the least significant bit (LSB) of each pixel value. We can modify the LSB in each pixel without making any noticeable difference. This way, an image can carry another secret image that is completely invisible to the naked eye.

What is a least significant bit?

Example: What is the LSB of 13? First convert the decimal representation to binary as shown above: 13 -> 1101.

The LSB is the **rightmost** digit, in this case a 1. The 3 LSBs are are the two rightmost digits, in this case 101, which is 5 in base 10.

2.1.2) Questions to Consider

Note: these are not part of the assignment, they are merely to help you think through the problem. If you get stuck, post on Piazza or come into office hours!

1. Given a number is divisible by 2, what would be the value of the LSB? Which of these binary numbers are divisible by 2: 1001, 10111, 10110, 111110?
2. How can we get the value of the LSB without converting a base 10 number into binary representation?
3. Would the binary number xxx00 (x can be either 0 or 1) be divisible by 4? If so why?
4. What would be the remainder of these binary numbers when divided by 4: xxx01, xxx10, xxx11
5. How can we extract the two LSBs of a base 10 number without converting it to binary?
6. How can we extract the `n` LSBs of a base 10 number without converting it to binary?

Please make sure you understand these questions before moving on. Understanding them will simplify your implementation of `extract_end_bits`.

2.2) Implementing `extract_end_bits`

In order to extract arbitrary numbers of these LSBs from pixel values, you will implement a helper function `extract_end_bits(num_end_bits, pixel)`. This function will output the **`num_end_bits` LSBs of `pixel` as an integer in base 10**.

Parameters:

- `num_end_bits`: the number of LSBs to return
- `pixel`: the pixel value whose LSBs will be returned

For example, going off of the example above, we could write:

```
extract_end_bits(1, 13) # get one LSB -> return 1
extract_end_bits(2, 13) # get two LSBs -> return 1
extract_end_bits(3, 13) # get three LSBs -> return 5
```

Hints:

- **Do not convert numbers into binary.**
- The implementation of this function should be **very short.**
- When thinking about how to implement `extract_end_bits`, the modulo (%) operator may be very useful.
- You can run **test_ps5_student.py** to check your implementation of extract_end_bits.

2.3) Recovering a binary image

In a binary (black & white) image, each pixel is represented by a numerical value representing its intensity. The minimum value for each pixel is 0 (which corresponds to black), while the maximum value (which corresponds to white) depends on the number of bits used for each pixel. In our examples, we will use 8-bit images. This means the pixels can take values between 0 and 255, inclusive (can you see why?).

In **ps5.py**, complete the black and white portion of the function `reveal_image` by implementing `reveal_bw_image` according to the docstring. This function takes in the filename of an image, uses the LSB value to find the hidden image, and returns it as a `PIL.Image` object.

Note: `reveal_image` should be able to handle both BW and RGB images; you'll implement the RGB portion in part c.

Important: The hidden image is embedded in the least significant bit and can be recovered using `extract_end_bits` with the appropriate `num_bits` argument on each pixel value. However, using the raw result from this operation will produce an image with very low contrast. Once you recover the information, you should carry out another operation to rescale the pixel values, so they can make use of the full range of available values.

In the problem set folder, you will find an image named **hidden1.bmp**. Using your function, find the secret image in this file and save a copy of it by using the PIL libary functionality. **Note:** for a grayscale image, the secret image should also be grayscale. Remember what you named the file as you will need the file name in a bit.

2.3.1) Questions to Consider

Note: these are not part of the assignment, they are merely to help you think through the problem. If you get stuck, post on Piazza or come into office hours!

1. What numbers in base 10 can be represented in 1 bit? In 2 bits?
2. What is the range of values of a BW pixel (or an element in a RGB pixel)?
3. How can we rescale LSB values to take advantage of a pixel's range?

2.4) Recovering a color image

The methods described above are robust enough to be applied to more bits. In this part, you'll work with an RGB image and use *three* least significant bits to recover the secret image.

In **ps5.py**, complete the relevant component of the function `reveal_image` by implementing `reveal_color_image` according to its docstring. For a color image, this function takes in the filename, extracts the secret image from the *three* least significant bits of each channel (red, green and blue) and returns it as a `PIL.Image` object. You will again need to rescale the pixels.

In the problem set folder, you will find an image named **hidden2.bmp**. Using your function, find the secret image in this file and save a copy of it by using the PIL libary functionality. Remember what you named the file as you will need the file name in a bit.

2.5) Making Your Art Your Own

At this point you should have three new images:

1. Filtered image_15.png
2. Unhidden hidden1.bmp
3. Unhidden hidden2.bmp

Using the provided helper function `draw_kerb`, run each of your images through this function to add your kerberos watermark to each one.

To get you started, if you had a image named `img1.png` and your kerberos was `timthebeaver` you could call `draw_kerb("img1.png", "timthebeaver)"` which would output an image in your directory named `img1_kerb.png`.

Once you have the three images with your kerb on them, combine them into **one** pdf for submission. You can use something like a pdf combiner to do this. You should be able to upload all three in their original file format and it will output one pdf file that you should submit.

NOTE: Before submitting your code to the website, comment out your `draw_kerb` function calls to avoid breaking the autograder!

3) Hand-in Procedure

3.1) Time and Collaboration Info

At the start of each file, in a comment, write down the names of your collaborators. For example:

```
# Problem Set 5
# Name: Jane Lee
# Collaborators: John Doe
```

Please estimate the number of hours you spent on the Problem Set in the question box below.

3.2) Half-way Submission

All students should submit their progress by the half-way due date (1 week before the final due date).
This submission will be worth 1 point out of the problem set grade and will not be graded for correctness. The intention is to make sure that you are making steady progress on the problem set as opposed to working on it in the final days before the due date.

You may upload new versions of each file until Nov 30 at 09:00PM. You cannot use extensions or late days on this submission.

Please refresh the page before submitting a new file. If you do not, your latest submission won't be updated.

Select File No file selected

Submit

You have infinitely many submissions remaining.

3.3) Submission

Be sure to run the student tester and make sure all the tests pass.

Submit both your python code and images pdf below.

Make sure you upload a .py file and a .pdf file to the submission boxes.

You may upload new versions until Dec 07 at 09:00PM, but anything uploaded after that time will be counted towards your late days, if you have any remaining. If you have no remaining late days, you will receive no credit for a late submission.

3.3.1) Python Submission

Select File No file selected

Submit

You have infinitely many submissions remaining.

3.3.2) Images PDF Submission

Select File No file selected

Submit

You have infinitely many submissions remaining.

6.100L Introduction to CS and Programming Using Python

Fall 2022