2022S1 CIIC3015 Project 2

Project information:

- Project 2 is due Nov 11 at midnight 11:59PM
- You may choose a partner **FROM YOUR SAME LAB SECTION**
    - If you prefer to do the project alone please select one of the **Single_#** groups
    - If you are having difficulty finding a partner for your project feel free to use **this spreadsheet** to try and find another unpaired student **in your section:**
- After your group submits your project you will have to book some time with your lab instructor to "defend" your work. **THIS IS WORTH 30% OF THE GRADE.** We will ask you questions about
    - what you did in the project
    - Why you did those things
    - What would happen if you changed some parts of the code

Image filters

Background:

Images are 2 dimensional arrays of the individual values (colors) for each pixel in the image. Each pixel has a location determined by the indices in the 2d array (list of lists) and takes on the color specified in the image file. Consider the image:

```
cross = [
    [0,1,0],
    [1,1,1],
    [0,1,0]
]
```

where 1 is white and 0 is black. This 2d list makes the image of a white cross in a black background. Recall how indexing works for nested lists: the cross variable is a 2d list so the 1st index will return an inner list item:

```
cross[0] == [0,1,0]
cross[1] == [1,1,1]
cross[2] == [0,1,0]
```

Then the 2nd index assumes you ALREADY got the item from cross at the correct position so you get an item from the inner list:

```
cross[0][0] == [0,1,0][0] == 0      cross[0][1] == [0,1,0][1] == 1      cross[0][2] == [0,1,0][2] == 0
cross[1][0] == [1,1,1][0] == 1      cross[1][1] == [1,1,1][1] == 1      cross[1][2] == [1,1,1][2] == 1
cross[2][0] == [0,1,0][0] == 0      cross[2][1] == [0,1,0][1] == 1      cross[2][2] == [0,1,0][2] == 0
```

Because of this we say that a pixel(y, x) is at row y and column x where (0,0) is the TOP LEFT of the image. That is to say larger y means farther DOWN in the image.

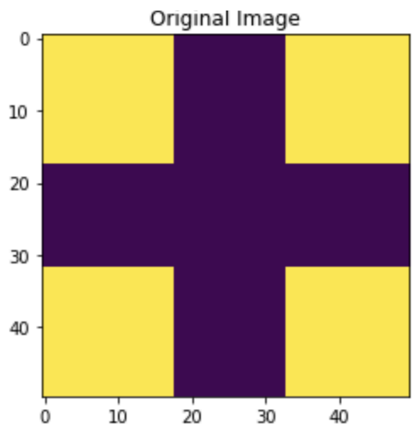
The following lines of code show an example of the libraries to visualize and open an imagen. You do not need to do this in your project. Since you assume your image is being saved in a 2D list.

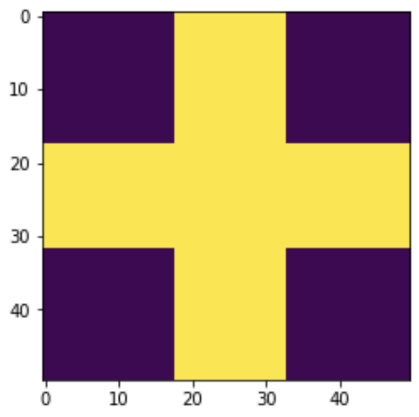
```
%matplotlib inline
import sys
from skimage import io
import matplotlib.pyplot as plt
from random import choice

logo = io.imread('images/plussign.pbm') # reads image as array of integers
logo = logo.tolist() # changes it to a python list
```

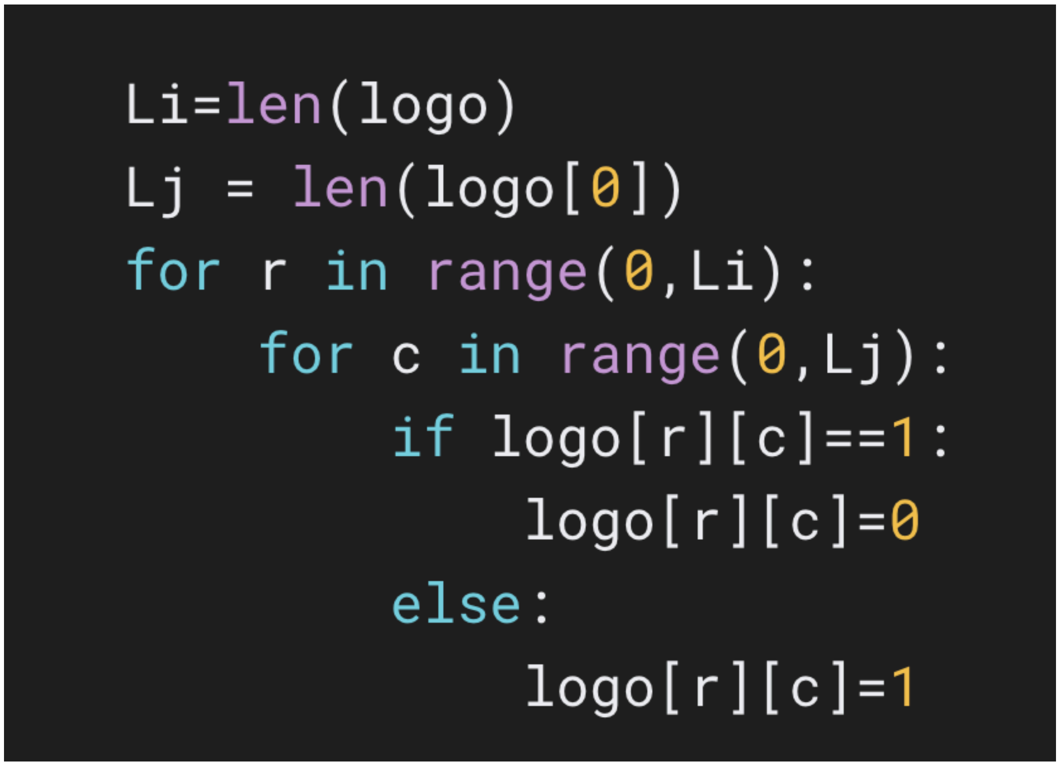
This image has 50 rows by 50 columns. Li is equal to 50

Inverted Image





This image contains only two values 0s and 1s (black and white). To invert the image, you need to switch the values as shown in the following code:



```
Li=len(logo)
Lj = len(logo[0])
for r in range(0,Li):
    for c in range(0,Lj):
        if logo[r][c]==1:
            logo[r][c]=0
        else:
            logo[r][c]=1
```

Images are not only black and white, of course, and there are different image formats for which the values inside the 2d list represent different things. One image format is grayscale which, instead of only being black and white, has different shades of gray. Grayscale often uses 8-bit resolution (remember binary?) which means that 0 is pure black, 255 is pure white, and everything in between is shades of gray where the higher the number the brighter/lighter it is and the lower the number the darker it is.

When we apply an image filter we are going through the image as a 2D list applying a filter function to each value (color/shade/pixel) in the image. Some such functions can be simple and act only on 1 pixel at a time such as increasing brightness (think adding a % of the color value to each pixel). Others are more involved and act on many pixels in the image at a time such as most instagram filters. In this project you will make your own image filters!

Task:

Part I:

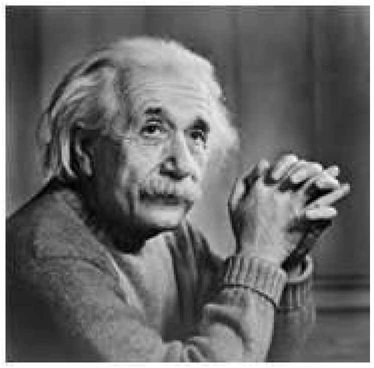
The first image filter you will be asked to create is a negative filter. The negative filter takes each value in the image and changes it for its INVERSE. That is to say, black goes to white and vice-versa. In the case of 0,1 black and white images this should be easy to see since you only have 2 options, but what does this mean for a grayscale image which has values from 0 to 255? We want to swap 0 for 255, 1 for 254, 2 for 253 and so on…

In practice, this means for every pixel we need to do the following:

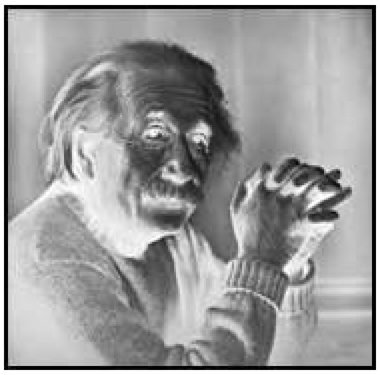
```
image[y][x] = 255 - image[y][x]
```

Write a function that takes in the image as a 2d list and returns a NEW 2d list representing the negative image. **DO NOT MODIFY THE INPUT IMAGE!!!**

Original:



Filtered:

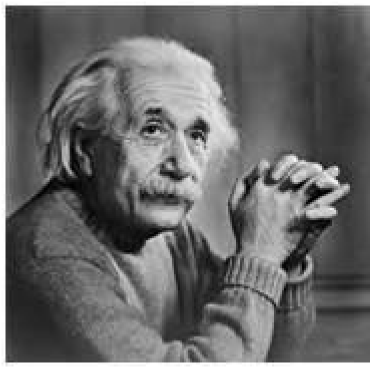


Part II:

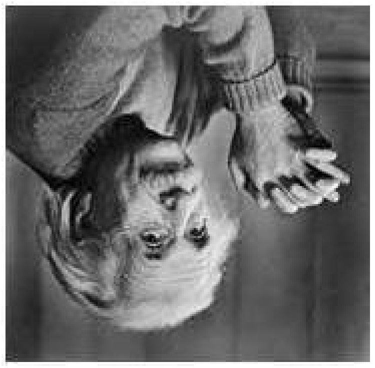
The next filter should flip the image vertically. This means each row of pixels needs to be swapped with a corresponding row from the other end of the image. That is to say, the 1st row should be swapped with the last row, the 2nd row is swapped with the second to last row, and so on...

Write a function that takes in the image as a 2d list and returns a NEW 2d list representing the flipped image. **DO NOT MODIFY THE INPUT IMAGE!!!**

Original:



Filtered:



Part III:

Background info:
The third filter should blur an image. This is a common filter used for smoothing out noisy images (images with a lot of random variation or low quality). The idea of a blur filter is that if a pixel is very different to its neighbors it's probably due to an image quality issue so the pixel value should be averaged out with the neighbors' values. If a pixel is very similar to its neighbors then the blur will still average it out but the pixel value will not change by much since every value in consideration will be similar.

Task:
A pixel's neighbors are all the pixels (including diagonals) which are next to the pixel so for pixel(y,x) in the center we would have the following:

```
image[y-1][x-1]      image[y-1][ x ]      image[y-1][x+1]
image[ y ][x-1]      image[ y ][ x ]      image[ y ][x+1]
image[y+1][x-1]      image[y+1][ x ]      image[y+1][x+1]
```

The template.py provided defines a 2d list called **`kernel`** which holds the *weights* of each pixel value when doing the blur filter. This ensures that each pixel's value is given higher importance than that of its neighbors since we want to respect the features of the image such as boundaries between objects. To calculate the new pixel value, the following *weighted average* of its and its neighbors pixel values must be calculated using the kernel. The weighted average for pixel(y, x) will be:

```
weighted_sum = (
    image[y-1][x-1] * kernel[0][0]    +    image[y-1][ x ] * kernel[0][1]    +    image[y-1][x+1] * kernel[0][2]    +
    image[ y ][x-1] * kernel[1][0]    +    image[ y ][ x ] * kernel[1][1]    +    image[ y ][x+1] * kernel[1][2]    +
    image[y+1][x-1] * kernel[2][0]    +    image[y+1][ x ] * kernel[2][1]    +    image[y+1][x+1] * kernel[2][2]  )

total_weights = (
    kernel[0][0]    +    kernel[0][1]    +    kernel[0][2]    +
    kernel[1][0]    +    kernel[1][1]    +    kernel[1][2]    +
    kernel[2][0]    +    kernel[2][1]    +    kernel[2][2]  )

weighted_average = weighted_sum // total_weights
```

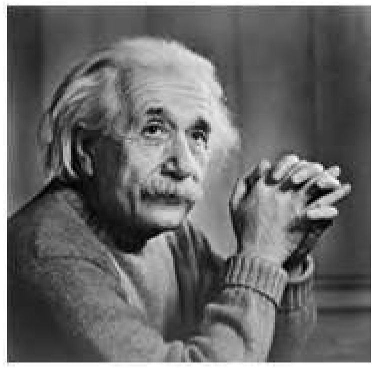
NOTE: A pixel(Y, X) is an ***edge*** pixel if at least one of the following is true:

1. Y == 0
2. Y == len(image) - 1
3. X == 0
4. X == len(image[y]) - 1

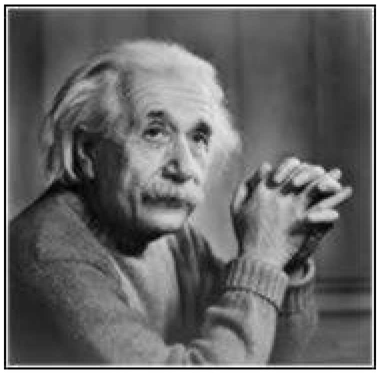
This weighted average formula will not work for the ***edges*** of the image since the edges don't have all the neighbors. **Your function should put a 0 for the edge pixels.**

Write a function that takes in the image as a 2d list and returns a NEW 2d list representing the blurred image. **DO NOT MODIFY THE INPUT IMAGE!!!**

Original:



Filtered:



Part IV Extra Credit:

NOTE:

This problem is for 3% extra credit which will be applied to your final grade in the class. That is to say, after we calculate your final grade if you do this problem you will get an extra 3% added.
Example: if your class grade is 87% and you did this problem then your final class grade will become 90%.
You will need to defend your solution to this problem just like the others so make sure you ***really*** understand your code!

The final filter is called tile and it creates a 2x2 tiled image based off of the input image. The tile filter will look very close to having 4 copies of the original image but each of the 4 tiles will be slightly different. Each tile is half the width and half the height of the original image. Picture this:

```
-------------------------------------------------------------
|                             |                             |
|                             |                             |
|       TILE (0, 0)           |         TILE (0, 1)         |
|                             |                             |
|_____________________________|_____________________________|
|                             |                             |
|                             |                             |
|       TILE (1, 0)           |         TILE (1, 1)         |
|                             |                             |
|_____________________________|_____________________________|
```

In order to make the tiles you will need to move certain pixels to their corresponding location in a tile. Consider a group of 4 pixels (2x2):

```
------------------------------------------------------------
|                             |                             |
|                             |                             |
|       pixel(i, j)           |       pixel(i, j+1)         |
|                             |                             |
|_____________________________|_____________________________|
|                             |                             |
|                             |                             |
|      pixel(i+1, j)          |       pixel(i+1, j+1)       |
|                             |                             |
|_____________________________|_____________________________|
```

then each pixel belongs to the tile in the same corner as they are: for example pixel(i, j+1) goes to Tile (0, 1). Notice that the tile indices are the same as what is added to the pixel indices. In each direction, every other pixel is being moved to a different tile, so there are half as many pixels in each tile. Consider tile(0, 0), the pixel (i, j) would need to move to the appropriate position considering the fact that half the pixels above and to the left have been "taken away" from tile (0, 0). This means that pixel(i, j) need to move to location (i // 2, j // 2). For example, consider the following pixels in the input image:

```
-------------------------------------------------------------------------------------------------------------------
|           X            |                        |           X            |                        |
|                        |                        |                        |                        |
|      pixel(0, 0)       |      pixel(0, 1)       |      pixel(0, 2)       |      pixel(0, 3)       |
|                        |                        |                        |                        |
|________________________|________________________|________________________|________________________|
|                        |                        |                        |                        |
|                        |                        |                        |                        |
|     pixel(1, 0)        |      pixel(1, 1)       |     pixel(1, 2)        |      pixel(1, 3)       |
|                        |                        |                        |                        |
|________________________|________________________|________________________|________________________|
|            X           |                        |           X            |                        |
|                        |                        |                        |                        |
|     pixel(2, 0)        |      pixel(2, 1)       |      pixel(2, 2)       |      pixel(2, 3)       |
|                        |                        |                        |                        |
|________________________|________________________|________________________|________________________|
|                        |                        |                        |                        |
|                        |                        |                        |                        |
|     pixel(3, 0)        |      pixel(3, 1)       |     pixel(3, 2)        |      pixel(3, 3)       |
|                        |                        |                        |                        |
|________________________|________________________|________________________|________________________|
```

In the new image they would become the following pixels located in **TILE(0,0)**:

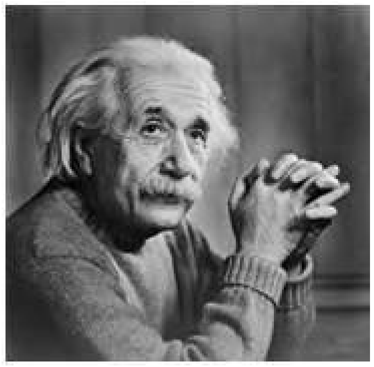
```
-----------------------------------------------------------
|                             |                            |
|  new_pixel(0, 0)            |   new_pixel(0, 1)          |
|                             |                            |
| original_pixel(0, 0)        | original_pixel(0, 2)       |
|_____________________________|____________________________|
|                             |                            |
|  new_pixel(1, 0)            |    new_pixel(1, 1)         |
|                             |                            |
| original_pixel(2, 0)        | original_pixel(2, 2)       |
|_____________________________|____________________________|
```

where new_pixel represents the location in the new image and original_pixel is the location of the pixel in the input image.

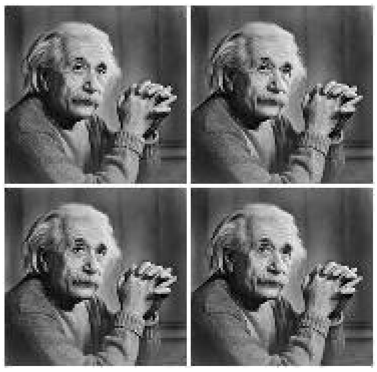
The question remains, how do the neighboring pixels get moved to each of their tiles? This is left for you to figure out but as a hint consider the distance between the top left pixel in each TILE. How far are they from TILE(0, 0) in the vertical direction? How about in the horizontal?

Write a function that takes in the image as a 2D list and returns a NEW 2D list representing the tiled image. **DO NOT MODIFY THE INPUT IMAGE!!!**

Original:



Filtered:

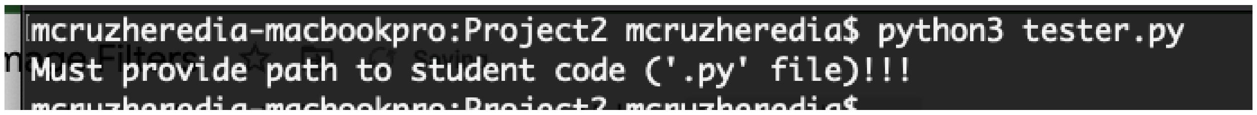


Testing your code:

DEMO VIDEO

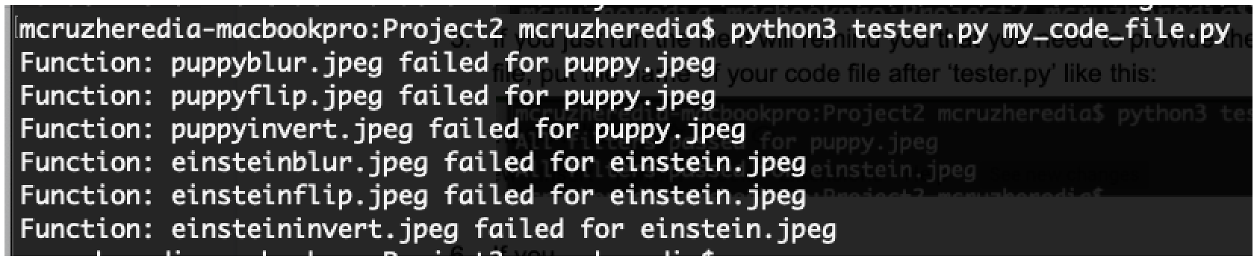
We have provided a ZIP file with all the files you need to test your code. If you extract the ZIP you should find before and after images for each filter. The original images are in the 'images' directory (folder) and the resulting images after applying the filters are in the 'expected_images' directory. You will also find a file called 'tester.py' to help you run your code on the original images and compare the results with the expected images (generated by Marco's solution). In order to run tester.py you will need to do the following:

1. Download the ZIP file
2. Extract it in a location (directory) of your choice
3. In the terminal you need to navigate to that location (directory)
    a. Windows
    b. Linux / Mac
4. Once you are in that directory, you need to run the file



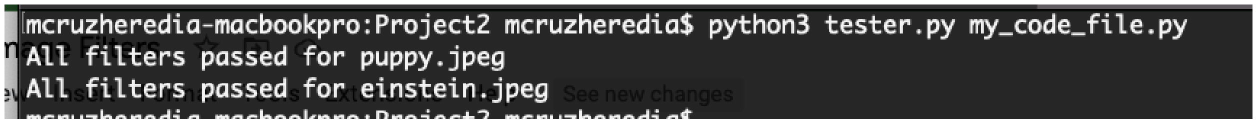
```
mcruzheredia-macbookpro:Project2 mcruzheredia$ python3 tester.py
Must provide path to student code ('.py' file)!!!
```

5. As you see above, if you just run the file it will remind you that you need to provide the `path` to your code file. If it's in the same directory this just means its name.
6. Put the name of your code file after 'tester.py' like this:



```
mcruzheredia-macbookpro:Project2 mcruzheredia$ python3 tester.py my_code_file.py
Function: puppyblur.jpeg failed for puppy.jpeg
Function: puppyflip.jpeg failed for puppy.jpeg
Function: puppyinvert.jpeg failed for puppy.jpeg
Function: einsteinblur.jpeg failed for einstein.jpeg
Function: einsteinflip.jpeg failed for einstein.jpeg
Function: einsteininvert.jpeg failed for einstein.jpeg
```

7. Once you correctly implement all the filters you will get the following message!



```
mcruzheredia-macbookpro:Project2 mcruzheredia$ python3 tester.py my_code_file.py
All filters passed for puppy.jpeg
All filters passed for einstein.jpeg
```

Note that you may need to install the following Python modules to run the tester:

- numpy
- pillow

If you have pip3 you can run the following command in the **terminal (or cmd)** to install:

```
pip3 install numpy pillow
```

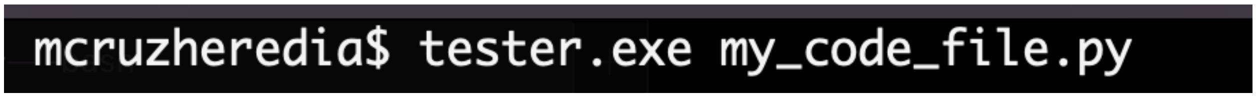
If not Google it 🙂

If windows is not cooperating:

FILE LINK

Here is a link to an *executable file* which you should be able to run on Windows. I have not really tested it since I do not have windows… Basically it is the same tester packaged with all the libraries (pillow, numpy, python3) needed to run it all in one file that Windows is supposed to be able to run.

Download the file and in CMD you should be able to run it like:



```
mcruzheredia$ tester.exe my_code_file.py
```

Of course the files have to be in the directory that the CMD is running the code or else CMD won't find the files.