PA6

PA 6

CSE 8A Fall 2021 PA 6

Due date: Tuesday, Nov 23 @ 11:59PM PST

(No late submission is allowed)

Provided Files

- `CSE8AImage.py` (DO NOT EDIT)

File(s) to edit and submit

- change_background.py (Task 1)
  - `def change_background(image, new_background, replace_color)`
  - `def color_distance(color1, color2)`
- flip.py (Task 2)
  - `def flip_vertical(image, col_index, row_index, height, width)`
  - `def check_validity(image, col_index, row_index, height, width)`
- pixelization.py (Extra Credit Task)
  - `def pixelization(image, square_size, column_percentage)`

⚠ Please note, in PA6, **there are hidden test cases**, which will not appear in your submission results. Also, tutors will grade your code manually. Therefore, the test case result you see when you submit your code is NOT final and passing all the test cases does not necessarily mean you can get full credit.

Part 0: Tips

- Ed has a strict limitation on how much memory your program can use and 2D lists are computationally expensive. If you ever run into situations where your program is not responsive or even killed, try to reduce the image size a little bit and see if this fix the issue. Feel free to make a private post if the above failed.

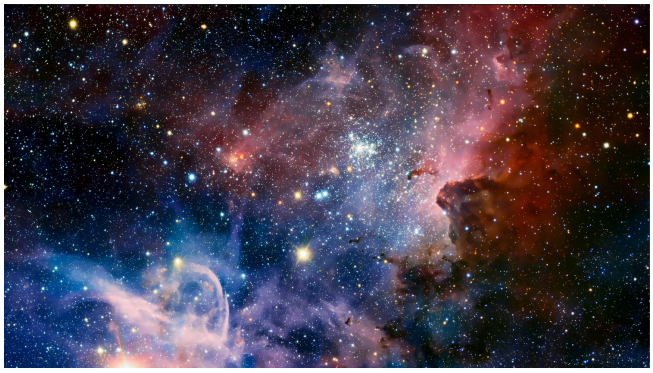
Part 1: Implementation

Background: Through Stepik readings, you learned about how to represent images and how to transform images. In this PA, you will gain some hands-on experience working with images by changing image background and flipping a certain region in the image.

Task 1: Change Image Background (23 points)

- In the Stepik reading, we have learned how to apply a red filter to an image. To further develop your understanding of how to manipulate images in python, now it's your turn to implement changing the image background.
- First of all, you need to go to the lab basement B240 and take a photo of yourself or together with your friends in front of the green screen. Feel free to look at the camera and do any pose you want. Feel free to refer to the one below as an example, but please do **NOT** use the following image. You should take your owns, and be creative :)

- Then you need to pick a background image that you like to replace the green screen in the original image. The background image is supposed to be bigger than the source image. Again, this is just an example and you shouldn't simply use this one for the PA.



- After you change the background, the new image will look something like the one below, albeit it isn't a perfect one. Feel free to play around with your functions a bit to eliminate as much green background as possible

- **YOUR TASK**: Implement two functions, and provide your custom image
  - `def change_background(image, new_background, replace_color)`
    - `image` is the source image (2D list of RGB tuples) that you take in front of the green screen
    - `new_background` is the background image that you want to use to replace the green screen
      - To make your life easier, you can assume both of the 2 arguments above are 2D list of tuples
    - `replace_color` is a tuple of RGB value that will be replaced in the source image. In our example, the replace_color is green. You need to decide the RGB value yourself. (More details below)
    - this function should return `None` and do manipulation directly on the argument `image`
  - `def color_distance(color1, color2)`
    - `color1` is the pixel color in the source image
    - `color2` is the replace_color (some green RGB value)
    - You need to calculate the distance between the two color using the following formula. Feel free to `import math` and use the `sqrt` function
      - $distance = \sqrt{(r1 - r2)^2 + (g1 - g2)^2 + (b1 - b2)^2}$.
  - After you've generated your output image, name it `creative.png` in your workspace. We'll download everyone's workspace and automatically take out that file from everyone. If you don't name it like so, it's very likely that your image won't be caught by the auto-fetching script and we will not be able to witness your creativity
- **More explanations:** What you need to do is, if the `color_distance` of the pixel color in the source image and the `replace_color` is within a threshold, you need to replace the source pixel color with the corresponding pixel color in `new_background`. **You** will decide the threshold! Feel free to come up with one from your intuition and run multiple experiments to see if the new image looks good enough.
- We will grade your code and look at your output pictures manually. We will have a vote for everyone's picture and give extra points to the top 10, so please unleash your creativity.

- You can use the methods provided in the `CSE8AImage.py` library to test your solution using the terminal. Useful functions include but are not limited to: `load_img()`, `save_img()`.
    - **Note that** you need to close and reopen the file when you save something new to the same file, otherwise the file content does not automatically refresh.

Task 2: Flip region (52 points)

`def flip_vertical(image, col_index, row_index, region_height, region_width)`
In Stepik, there's an example of flip left to right. But this time we want to select a specific rectangular region of the image and flip the region **top to bottom**.

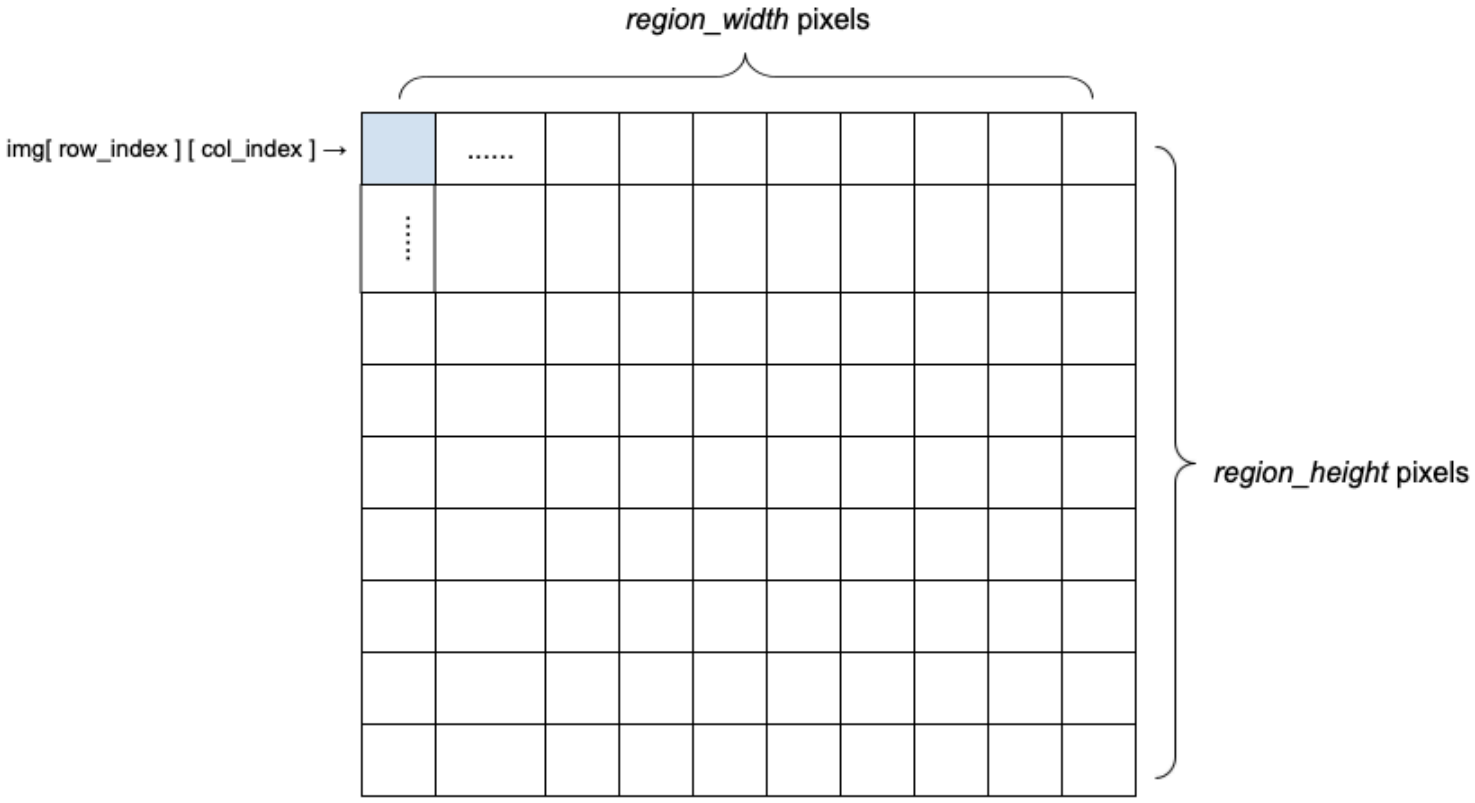
- input:
    - `image`: 2D list of tuples representing RGB values
    - `col_index`: an integer value that represents the starting column of the pixel at the upper-left corner of the selected rectangular region
    - `row_index`: an integer value that represents the starting row of the pixel at the upper-left corner of the selected rectangular region
    - `region_width`: the width of the rectangular region that we want to flip
    - `region_height`: the height of the rectangular region that we want to flip
- output:
    - return `False` if parameters are not valid and no operations would be done to the image
    - return `True` if the image is flipped successfully

`def check_validity(image, col_index, row_index, region_height, region_width)`

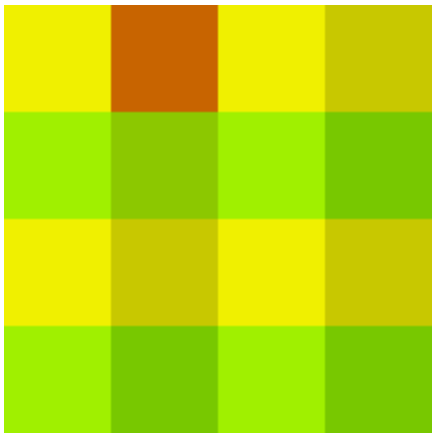
You would implement another function to check if input parameters are valid. Input parameters are the same as above, and you need to return `True` if parameters are valid, and `False` otherwise. Rules for valid parameters are as follow:

1. `col_index, row_index, region_height, region_width` should all be greater than or equal to 0
2. The rectangular region specified by the parameter should be **entirely inside** the image
3. You can assume `image` is always a valid 2D list of tuples and we don't test on this edge case

The specified flipping region is illustrated below:

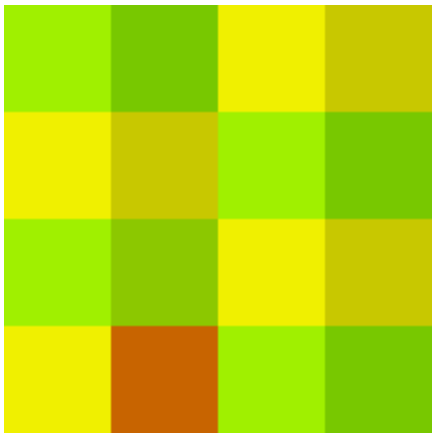


[Example 1]:
`image` =
[[(240, 240, 0), (200, 100, 0), (240, 240, 0), (200, 200, 0)],
[(160, 240, 0), (140, 200, 0), (160, 240, 0), (120, 200, 0)],
[(240, 240, 0), (200, 200, 0), (240, 240, 0), (200, 200, 0)],
[(160, 240, 0), (120, 200, 0), (160, 240, 0), (120, 200, 0)]]



If we call `flip_vertical(img,0,0,4,2)`,
`image` becomes
[[**(160, 240, 0), (120, 200, 0)**, (240, 240, 0), (200, 200, 0)],
[**(240, 240, 0), (200, 200, 0)**, (160, 240, 0), (120, 200, 0)],
[**(160, 240, 0), (140, 200, 0)**, (240, 240, 0), (200, 200, 0)],

[**(240, 240, 0), (200, 100, 0)**, (160, 240, 0), (120, 200, 0)]]
The bolded region is the region selected by this function call.



[Example 2]

Original image (200 x 300)

If we do following steps:

- `image = load_img('input.png')`
- `flip_vertical(image, 100, 60, 120, 120)`
- `save_img(image, 'flipped.png')`

We got the image below

Extra Credit: Pixelization Filter (10 Points):

Your task:

- Write a simple pixelization method that pixelates an image by averaging R, G, B values of `square_size x square_size` sized grids respectively.
- Implement two functions as described below

`def pixelization(image, square_size, column_percentage)`

- Input:
  - `image`: 2D list of tuples representing RGB values, you may assume this is a valid 2D list of tuples.
  - `square_size`: an integer indicating the `square_size x square_size` which we will pixelate the region by.
    - e.g. `square_size = 2`, pixelization region = 320x400, every non-overlapping 2x2 region will have the same color (RGB values) after pixelization
    - You may assume `square_size` >= 1
  - `column_percentage`: an integer between `[0,100]` indicating how much of the `image` should be pixelated on the **right-hand side**.
    - e.g. `image` is 400x400 (width x height), if `column_percentage = 50`, then pixelization should only be done to the 200x400 pixels on the right-hand side.
    - if `column_percentage = 80`, then pixelization should be done to the 320x400 pixels on the right-hand side. $(\frac{80}{100}) * 400 = 320$.
    - Note: if `column_percentage = 100`, the whole `image` should be pixelated. Similarly, if `column_percentage = 0`, no changes will be made to `image`.

[Note]:

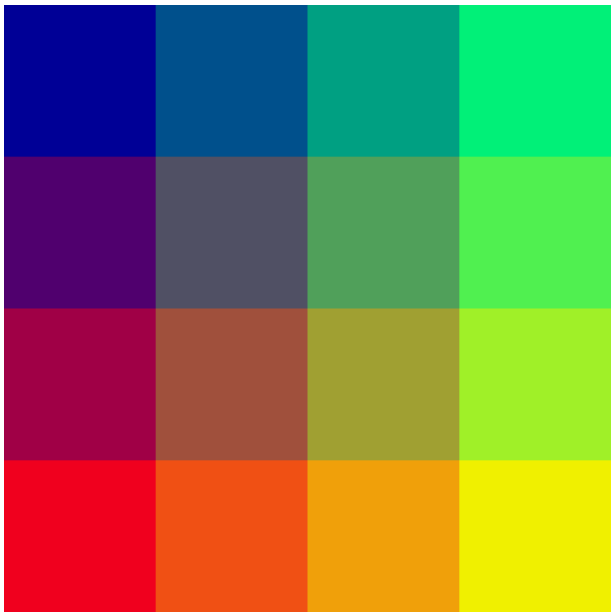
1. You may assume that the width and height will be divisible by `square_size` for region to be pixelated
2. You may assume that `(1 - column_percentage/100) * image width` will be an integer.
3. Pixelization should only be applied to the `column_percentage` % on the **righ-hand side**.

- Output:
  - No output, `image` will be updated

[Example 1]

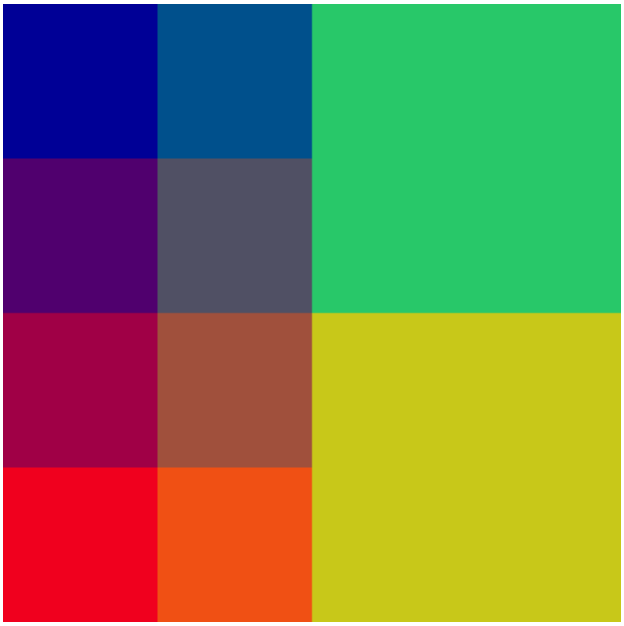
`image` =

[[(0, 0, 150), (0, 80, 140), (0, 160, 130), (0, 240, 120)],
[(80, 0, 110), (80, 80, 100), (80, 160, 90), (80, 240, 80)],
[(160, 0, 70), (160, 80, 60), (160, 160, 50), (160, 240, 40)],
[(240, 0, 30), (240, 80, 20), (240, 160, 10), (240, 240, 0)]]



If we apply `pixelization(image, 2, 50)`,
`image` would become
[[(0, 0, 150), (0, 80, 140), **(40, 200, 105), (40, 200, 105)**],
[(80, 0, 110), (80, 80, 100), **(40, 200, 105), (40, 200, 105)**],
[(160, 0, 70), (160, 80, 60), **(200, 200, 25), (200, 200, 25)**],
[(240, 0, 30), (240, 80, 20), **(200, 200, 25), (200, 200, 25)**]]

Red value for top right: $(0 + 0 + 80 + 80)/(2 * 2) = 40$
Green value: $(160 + 160 + 240 + 240)/(2 * 2) = 200$
Blue value: $(130 + 120 + 90 + 80)/(2 * 2) = 105$
Therefore, the top right 2x2 becomes (40, 200, 105).



[Example 2]

Original image (200 x 300)

If we do following steps:

- `image = load_img('input.png')`
- `pixelization(image, 10, 60)`
- `save_img(image, 'pixelized.png')`

We got the image below

Part 2: Style (5 points)

Coding style is an important part of ensuring readability and maintainability of your code. We will grade your coding style in all submitted code files (**including optional extra credit files if submitted**) according to the style guidelines. You can keep these guidelines in mind as you write your code or go back and fix your code at the end. For now, we will mainly be focusing on the following:

- **Use of descriptive variable names**: The names of your variables should describe the data they hold. Your variable names should be words (or abbreviations), not single letters.
  - e.g. `a` --> `index_of_apple`; `letter1` and `letter2` --> `lower_case_letter` and `upper_case_letter`

    - Exception: If it is a loop index like `i`, `j`, `k`, one char is OK and sometimes preferred.
- **Inline Comments**: If there is a length of code that is left unexplained, take the time to type a non-redundant line summarizing this length of code (e.g. `#initialize an int` is redundant, vs. `#set initial length to 10 inches`). It will let others who look at your code understand what's going on without having to spend time understanding your logic first. But don't be too descriptive, as too many comments reduces readability.

Part 3: Conceptual Questions (20 points)

You are required to complete the Conceptual Questions in PA lesson. There is no time limit but you must sumit by the PA deadline. You can submit multiple times before the deadline. Your latest submission will be graded.

Submission

Turning in your code

Submit all of the following files to PA Lesson on EdStem via the "Mark" Button by **Tuesday, Nov 23 @ 11:59PM PST (No late submission is allowed**):

- `change_background.py`
- `flip.py`
- `pixelization.py`

Evaluation

- **Correctness** (75 points **excluding** 10 points extra credit for Pixelization Filter)
  You will earn points based on the autograder tests that your code passes. If the autograder tests are not able to run (e.g., your code does not compile or it does not match the specifications in this writeup), you may not earn credit. Your latest submission will be graded.
- **Style** (5 points)
- **Conceptual Questions** (20 points)

PA6 concept questions

Question 1 *Submitted Nov 23rd 2021 at 2:23:16 pm*

What will happen when we run this code below?

```
loc = (12.83, 11.93)
a = loc[0]
b = loc[1]
b = 0
b = a
print(loc)
```

- [x] (12.83, 11.93) will be printed
- [ ] (12.83, 0) will be printed
- [ ] (0,0) will be printed
- [ ] (12.83, 12.83) will be printed
- [ ] Error: Tuples are immutable

Question 2 *Submitted Nov 23rd 2021 at 2:28:05 pm*

Type conversions are commonly used in Python programming. Please look at the statement below and **select all the statements that are correct**.

```
x = 20                  #statement 1
price_tag = str(x)      #statement 2
money = float(price_tag)#statement 3
```

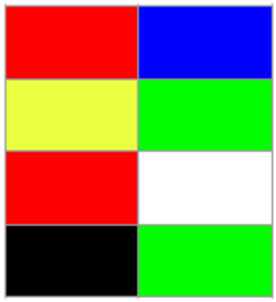
- [x] money is a float type variable
- [ ] x is int variable throughout the execution of the three lines of the code
- [ ] statement 3 causes an error when executed

- [x] price_tag is a str variable and ends up being a float type variable
- [x] variable x starts as an int variable and ends up being a float type variable

Question 3 *Submitted Nov 23rd 2021 at 2:28:39 pm*

Given the picture below, and assume that variable img points to the list of tuples reprenting this picture, what is img[1][1]

(Note: Colors from left to right, top to bottom: [[red, blue], [yellow, green], [red, white], [black, green]].)



- ( ) (255, 0, 0)
- ( ) (0, 0, 255)
- (•) (0, 255,0)
- ( ) (255, 255, 0)

Question 4 *Submitted Nov 23rd 2021 at 2:32:10 pm*

Select all the statements that are true for the following code.

```
img = [[(0,0,0), (255,255,255)],
       [(255,0,0), (0,0,255)],
       [(0,255,0), (100,100,100)]]
img[0] = img[1]
img[1][1] = (90, 90, 90)
print(img[0][1])
#codes that may be added here for choices C and D
```

- [x] It will print out (90, 90, 90)
- [x] img[0] and img[1] are referring to the same row after the code executes.
- [ ] The img 2D list will have 4 rows and 2 columns after the current code executes.
- [ ] if I put img[0][1] = (10, 20, 30) after the print statement, and then I add in a print statement, print(img[1][1]), it will print out (90, 90, 90)
- [x] if I put img[0][1] = (10, 20, 30) after the print statement, and then I add in a print statement, print(img[1][1]), it will print out (10, 20, 30)