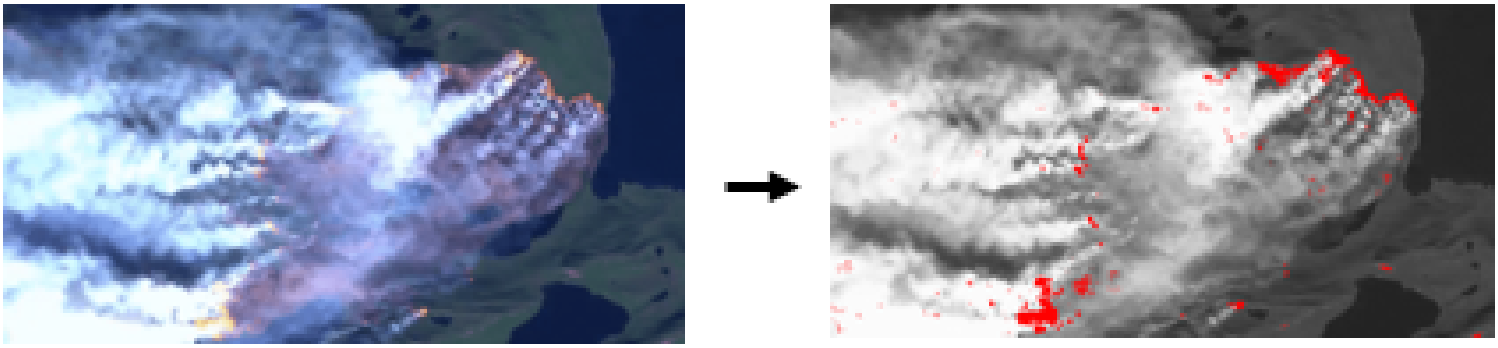
Assignment 3: Images

Assignment 3: Images

In this assignment we are going to be writing image algorithms! Woot.



We are going to use a library we put together for you called SimpleImage. You can read all about it in this handout:

https://codeinplace2021.github.io/pythonreader/en/images/

Q1: Code in Place Filter

Write a program that asks the user to enter an image file, loads that file and applies the “Code in Place” filter.

To apply the Code in Place filter, you are going to change every **pixel** to have the following new red, green and blue values, based off the pixels old red, green and blue values:

```
new red value = old red value * 1.5
new green value = old green value * 0.7
new blue value = old blue value * 1.5
```

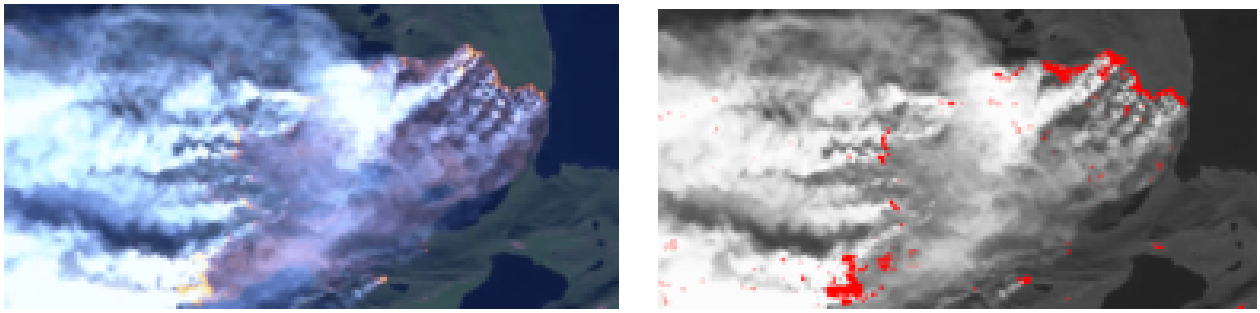
Problem written by Chris Piech. Inspired by image library and examples from Nick Parlante.

Q2: Finding Forest Flames

We're going to start by writing a function called `find_flames` (in the file `forest_fire.py`) that highlights the areas where a forest fire is active. You're given a satellite image of Greenland's 2017 fires (photo credit: Stef Lhermitte, Delft University of Technology). Your job is to detect all of the "sufficiently red" pixels in the image, which are indicative of where fires are burning in the image. As we did in class with the "redscreening" example, we consider a pixel "sufficiently red" if its red value is greater than or equal to the average of the pixel's three RGB values times a constant INTENSITY_THRESHOLD.

Recall that the average of a pixel, which has red, green and blue values is:

```
average = (red + green + blue) / 3
```



Original forest fire image on left, and highlighted version of image on right.

When you detect a "sufficiently red" pixel in the original image, you set its red value to 255 and its green and blue values to 0. This will highlight the pixel by making it entirely red. For all other pixels (i.e., those that are not "sufficiently red"), you should convert them to their grayscale equivalent, so that we can more easily see where the fire is originating from. You can grayscale a pixel by summing together its red, green, and blue values and dividing by three (finding the average), and then setting the pixel's red, green, and blue values to all have this same "average" value.

Once you highlight the areas that are on fire in the image (and greyscale all the remaining pixels), you should see an image like that shown on the right in the figure. On the left side of the example image, we should the original image for comparison.

Note: to make this algorithm work on different images of fire, select an appropriate INTENSITY_THRESHOLD value.

Problem written by Sonja Johnson-Yu.

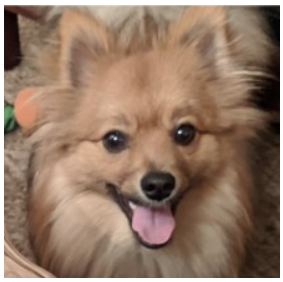
Q3 (optional; medium): Reflection

Write a function that returns an output image that is twice the height of the original. The top half of the output image should be identical to the original image. The bottom half, however, should look like a reflection of the top half. The highest row in the top half should be "reflected" to be the lowest row in the bottom half. This results in a cool effect.

Problem written by multiple instructors. We are seeking out the photographer who took this picture.

Q4 (optional; hard): Warhol Effect

Write an algorithm that takes in a square patch like this photo of Simba the Dog:



And creates an image which has the patch copied 6 times (in 2 rows and 3 columns) where each patch gets re-colored. This effect is inspired by some of Andy Warhol's paintings.

We strongly recommend implementing a function, like so:

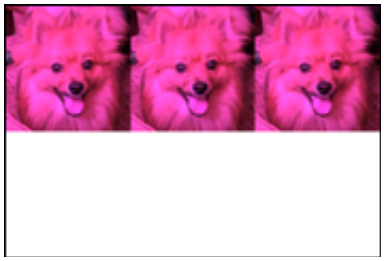
```
def make_recolored_patch(red_scale, green_scale, blue_scale):
```

Which returns a new colored patch. See the starter code for details.

Don't try to match the colors in this example exactly. Experiment with different combinations of red_scale, green_scale and blue_scale. The pink Simba was generated by

```
make_recolored_patch(1.5, 0, 1.5)
```

A few milestones:



Define other functions too! How about a function which adds a colored patch to the final_image at a given row, column?

Problem written by Chris Piech. Inspired by a problem called "Quilt" by Julie Zelenski.

Warhol Effect Milestones

To help you take the Warhol Effect problem one step at a time, we've written some structured milestones for you to follow.

Pseudocode / roadmap

When coding a complicated task, it's frequently helpful to think of the problem in your native language first, instead of diving straight into Python. How would you explain how to solve the problem to a child? How might you split the task into smaller subtasks? Here's our pseudocode for this problem, with the corresponding milestones:

- The Warhol Effect is characterized by 6 patches of the same image recolored in different ways. This sounds like a good place to decompose a helper function with parameters, since we're doing the same task (recoloring a patch) over and over, but need to be able to customize *how* we're doing the task (we want to recolor in different ways for each patch to create the Warhol Effect!). Milestone 1, `make_recolored_patch(red_scale, green_scale, blue_scale)` , will implement this.
- Once we know how to make recolored patches, we can get started arranging them on the canvas. We'll need to figure out how to place a single patch onto the canvas at a given position. Milestone 2, `put_patch(final_image, start_x, start_y, patch)` , will implement this.
- After we've written code to place the patch wherever we'd like, we'll start by putting 3 patches in a row onto the final image. Milestone 3 will implement this.
- After we've put one row onto the final image, we can extend our code to put all the necessary rows (in this case, 2) onto the final image. Milestone 4 will implement this.
- At this point, we're pretty much done with the problem, and can add all the creative we like by changing the parameters to `make_recolored_patch(red_scale, green_scale, blue_scale)` to generate differently colored patches!

Hopefully this thought process makes some sense to you -- happy coding! Below are the milestones in more detail.

Milestone 1: `make_recolored_patch(red_scale, green_scale, blue_scale)`

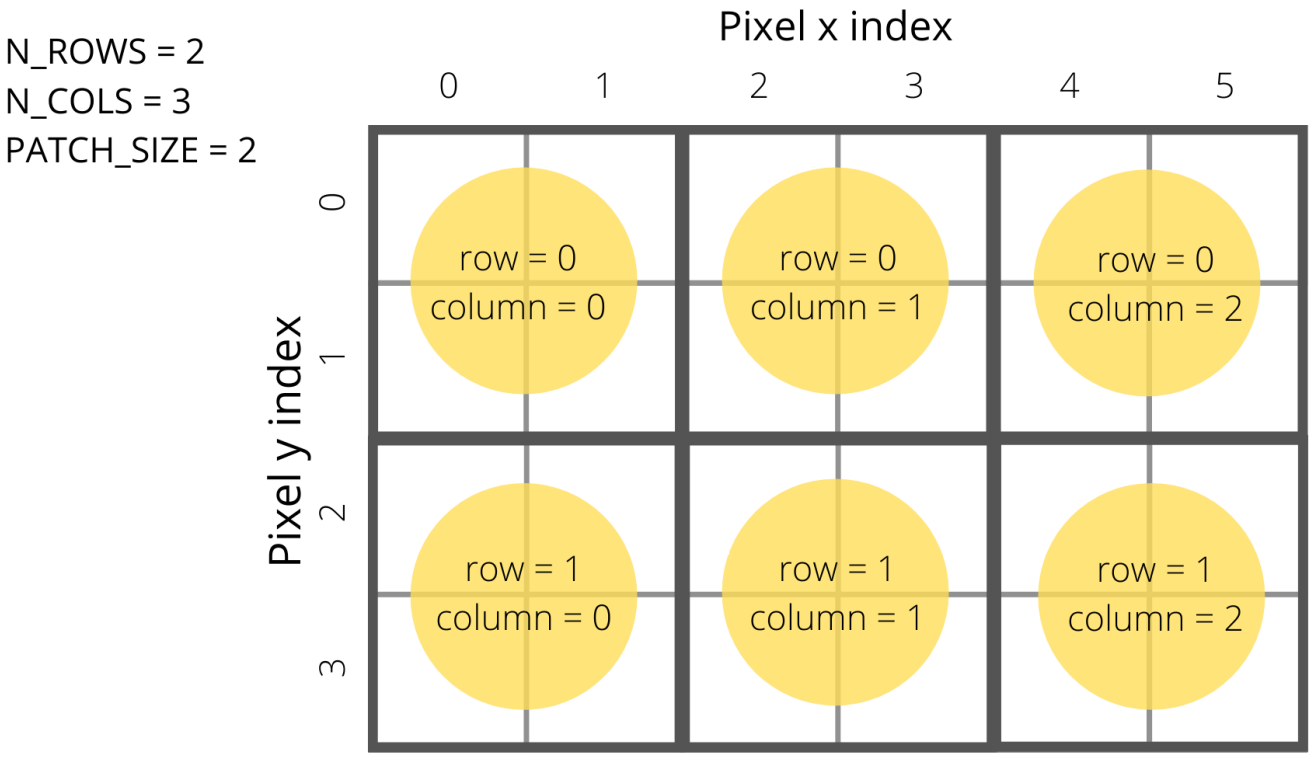
As alluded to in the problem description, we recommend you start with a helper function that takes multipliers for the red, green, and blue values of that patch. You'll want to create a new `SimpleImage` from the `PATCH_NAME` and, similar to the Code in Place Filter problem earlier in this assignment, modify every pixel in the patch and then return it.

You can test that your patch has been recolored successfully by replacing the line `final_image.show()` at the end of `main()` with `patch.show()` .

Milestone 2: `put_patch(final_image, start_x, start_y, patch)`

The next step we suggest is to implement a helper function that will place a patch onto the final image with its top left corner at (`start_x` , `start_y`).

Here's a helpful diagram (in general, for image problems, we highly recommend you draw pictures!), where for the sake of simplicity we're pretending each yellow circle is a patch that is 2 pixels wide and 2 pixels tall (`PATCH_SIZE = 2`):



Let's say we're looking at the second patch in the first row in this diagram. In this case, `start_x` is 2 (which, not entirely by accident, happens to be the value of `PATCH_SIZE`), and `start_y` is 0. You'll need to figure out how to use `start_x` and `start_y` in an expression in order to set the correct pixels in `final_image` to the corresponding pixels in `patch` ! It may help to draw out a similar diagram to the above for just one patch.

You can test this works by calling this function once and placing a patch somewhere on `final_image` before moving on.

Milestone 3: do one row

Now that you've got a helper function to put one patch in the correct place, try calling `put_patch(final_image, start_x, start_y, patch)` to place the first row of three patches. Reference the above diagram again, and pay attention to the pattern in the values of `start_x` as we go from the left patch to the one in the middle to the one on the right. It may be helpful to quickly redraw the diagram for yourself but add in the values of `start_x` and `start_y` for each patch.

Milestone 4: do all rows

Now that you can do one row, all that's left is to do the next one! This time, pay attention to how the

value of `start_y` changes from the top row to the bottom row.

Milestone 5: finishing touches!

At this point, you've successfully placed all 6 patches in their correct positions, which is by far the hardest part of this problem! From here on out, you have full creative reign -- use `make_recolored_patch()` with whatever parameters you want for each patch, to make your own Warhol-inspired creation. Some fun ideas for inspiration:

- Manually pick 6 "filters" you like -- can you hardcode them in?
- Can you do the transformations mathematically, using the row / column numbers for each patch, so that the patches are all different and you don't hardcode any numbers?
- Can you do the transformations using randomness?
- How might you write a helper function like `get_rgb_scales(row, col)` to return `red_scale`, `green_scale`, and `blue_scale` values? (hint: lists)

Q5 (optional): Extensions

The joy of programming is often making something that you is your own. Create any image algorithms you like.

Problem written by you!

Clarification Questions