Assignment #3: Fun with Images

On-time deadline: 11:59 PM on Thursday, July 18
Extended deadline: 11:59 PM on Friday, July 19
This assignment should be done individually.

This assignment consists of a set of warmups and three larger problems. You can download the starter code for this project under the “Assignments” tab on the CS106AP website. The starter project will provide Python files for you to write your programs in.

This assignment will focus on different applications of image processing, which is a powerful tool used in medicine, computer vision, photography, and many other fields. We’ll get practice with double for loops, pixel manipulation, lists, and string parsing. Image processing is an area in which computers allow us to tackle problems that we otherwise couldn’t. (Imagine updating millions of pixels by hand!)

For this assignment, you will have to run all of the programs from the command line. Recall this means using the terminal in PyCharm and running programs via the command format discussed in Lecture 11. There will be no run configurations or doctest configurations packaged with the assignment. To run doctests, you should right-click on the doctests in the function you want to run and select the ‘Run’ option from the corresponding menu.

The handout for this assignment is on the longer side, but don’t be intimidated by its length! It is long because it’s designed to give you step-by-step instructions for implementing several cool algorithms!

AN IMPORTANT NOTE ON TESTING:

For each problem, we give you specific guidelines on how to begin decomposing your solution. While you can add additional functions for decomposition, you should not change any of the function names or parameter requirements that we already provide to you in the starter code. Since we include doctests for these pre-decomposed functions, editing the function headers can cause existing tests to fail.

We are only requiring you to write doctests for some of the functions in Part 3 (Parse Mystery). As a requirement for this part, you must adhere to the following testing guidelines:

- For each pre-decomposed function that we provide you, you must write at least 3 additional doctests per function.

All tests for a given function should cover completely separate cases, and we encourage you to consider both common use cases and edge cases as you write your doctests. Using good testing practices and thinking through possible inputs/outputs for your code will help increase your likelihood of getting full functionality credit for your work!

Based on handouts and problems by Nick Parlante and John Nicholson (nifty.stanford.edu).
Setup: Installing Pillow

Before you get started on the assignment, make sure that you have run through the Pillow installation instructions, which can be found here. If you cannot get Pillow installed successfully, please come to LaIR or Kylie, Sonja, or Nick’s office hours, listed on the CS106AP home page.

Part 0: Warmups (warmups.py)

To get started, let’s do a couple of warm-up problems to practice looping over images and manipulating them.

1. We’re going to start by writing a function 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 red pixels. Recall that we consider a pixel “red” if its red value is greater than the average of the pixel’s three RGB values times some hurdle factor. In this case, we have provided you with an appropriate hurdle factor of 1.05 via a constant named HURDLE_FACTOR in warmups.py. Once you’ve detected a red pixel, set its red value to 255 and its green/blue values to 0.

For all other non-red pixels, convert them to grayscale, so that we can more easily see where the fire is originating from. Recall that it’s possible to 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 RGB values to all equal this “average” value.

Once you highlight the areas that are on fire, you should see an image as shown in Figure 1.

![Figure 1: Forest fire image before (left) and after (right) being highlighted](image)

2. 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, seen below in Figure 2!

Based on handouts and problems by Nick Parlante and John Nicholson (nifty.stanford.edu).
Part 1: Bluescreen Challenge (bluescreen.py)

We're going to walk you through a couple of steps so that you can create your own bluescreen image! Real greenscreens/bluescreens are used in movies and weather reports in order to place a foreground image onto a background (you can learn more about them here!). You're going to have the chance to code up a bluescreen algorithm and then get creative! Steps 1 to 5 below take you through how to get the necessary images for this challenge, and step 6 outlines the details of the challenge.

1. Take several foreground images

First, create several "foreground" pictures with people (or anything else!) in front of a solid-colored background, and save those images on your computer. To aid you with this, we will have bluescreens set up in class each day through Wednesday, July 17. You can take the pictures in class or on your own. We recommend that you take several different foreground images so you have multiple images to experiment with. For example, take one image with people in the center and another with them off to the side to fit in with interesting parts of the background. You can also try dressing the people in the foreground with blue clothes, so the background image will show through those areas. You don't have to use blue—green is also popular. Note that blue tarps don't usually work very well as a background, because they tend to have shiny white areas that lack hue. You might be able to fix this in "post" below.

A note on image size

Digital cameras create very large images like 4000 by 3000 pixels (12 megapixels!), and such large images will take too long to process and have far more pixels than the displays we have. Keep your original images, and create smaller versions for processing, reduced to a width of 400 or 500 pixels with a filename like "myimage-400.jpg". You'll need to find a balance between finding an image size...
that looks good but doesn’t take too long to process. A good ballpark estimate is about 1 or 2 seconds.

On the Mac, the built-in Preview application can crop and resize and export images, and on Windows, Paint and Windows Photo Editor function similarly.

2. Fix It In Post (optional)

"Fix It In Post" is a sort of Hollywood joke, referring to fixing some on-camera mistake later on in the computer post-production phase. This is totally optional, but you can do the same sort of thing with your bluescreen image. For example, if part of the blue background is not quite big enough, you can edit the image in a paint program, and draw a blue rectangle over the area to just make it be blue the way you want. Since the blue gets removed by the bluescreen algorithm, it never appears in the final image. By adding blue to the foreground image, you’re creating a “hole” where the background image will show through. On the Mac, the built-in Preview image app has some ability to put colored rectangles and ovals on images, and on Windows the Paint program is similar.

3. Background image size

Your bluescreen algorithm will loop over all of the pixels in the image and at times will access the corresponding (x, y) pixel in the back image. There's a possible problem: what if the background image is smaller than the foreground image? In that case, the code could try to access an (x, y) that exists in the foreground, but does not exist in the background. In that case, SimpleImage will throw an error like this:

Exception: bad coordinate x 600 y 600 (vs. image width 500 height 407)

There are two solutions to this, both of which are valid:

- Manually make sure that your background image is at least as big as your foreground image.

- SimpleImage includes a function called `make_as_big_as()` for just this case:

  ```
  background.make_as_big_as(image)
  ```

  This function scales up the size of `background` if necessary, so that it is as wide and as tall as `image`. Using this function, your code can scale the background image up to the necessary size on the fly. It is important to note that this function modifies `background` in place and does not return a new image.

---

Based on handouts and problems by Nick Parlante and John Nicholson (nifty.stanford.edu).
4. Upload your images

To use images in your bluescreen program, you'll need to upload the image files to the `images` folder in the Assignment3 directory. You can do this using Finder on a Mac or Windows Explorer on a PC or by dragging the image file directly into the PyCharm project.

You can upload any number of files to use as foreground or background images in your code. Filenames should not include non-alphanumeric characters (besides `-'`, `'_`, and `'`) and should end in "jpg" or "jpeg" or "png." Please limit your file size to 500K. If you store an image called “blue.jpg” in the `images` folder, then to access the image in your code, the filename should be “images/blue.jpg.”

5. Saving your output

When your code displays an image that you want to keep, you can save it using the photo editor in which the image automatically opens. Alternately, you can take a screenshot and later crop it down to your preferred size. You will need to save the image somewhere that is easily accessible (the `images` folder in the Assignment3 directory is a good choice), so that you don’t forget to upload it to Paperless!

6. Custom image exercises

You will create 2 custom bluescreen images. For each image, you will turn in the final output image and the code that created it. Ideally, you should be able to share the bluescreen code between the first and second custom bluescreens, though you will need to use different filenames (and potentially different heights/widths). Think about what common tasks are shared in the process of creating both images and decompose functions to do these tasks.

On Tuesday, July 23 we will celebrate being done with the midterm by having an in-class art show using the images you produce with your bluescreens (so don't turn in anything you wouldn't want the entire class to see, in person and on video!).

The challenge categories for the images will be:

- Most artistic
- Most humorous
- Best use of background
- Best image of you hanging out with someone super famous

The creators of the images that win the popular vote in each of these categories will win a special prize!
Custom Bluescreen #1
The first bluescreen image produced must include yourself in the foreground and must use your bluescreen processing code. Everything else is up to you! When you have your final image, save it as “bluescreen-1.png.”

Custom Bluescreen #2
For the second bluescreen image, you may use any foreground image you like. The foreground should not be a CS106AP stock image, and not the same image as the first custom bluescreen. When you have your final image, save it as “bluescreen-2.png.”

Files to submit: bluescreen.py, bluescreen-1.png, bluescreen-2.png

Part 2: Ghost (ghost.py)
Suppose we are trying to take a picture of Stanford, but each image has people walking through the scene, as shown in Figure 3:

![Figure 3](image)

Figure 3: Three images of Hoover Tower, with pedestrians walking through.

We’d like to look through all these images and figure out a way to “ghost” out all the people and make them disappear, giving us a clear view of Hoover tower!

![Figure 4](image)

Figure 4: A sample output of your program in ghost.py.

Ghost algorithm motivation

Let’s say that we have 3 images. From these images, consider the three pixels that exist at coordinate \((x, y)\) in each respective image. Most of the time, these three pixels will look the same across all three pictures, but in some cases, one of the pixels will look different.

Based on handouts and problems by Nick Parlante and John Nicholson (nifty.stanford.edu).
because there’s a person at that location in that image. We’ll call pixels where there’s a person “outlier” pixels, because they are different from all the other pixels that exist at that location in the other photos. We’ll assume that outlier pixels are always in the minority – that is, there will always be more normal pixels than outliers.

For example, suppose the three pixels at $x=100, y=50$ in each of our three images look like this (we use the shorthand $(\text{red}, \text{green}, \text{blue})$ to refer to the RGB values):

\begin{align*}
\text{First photo: } (1, 1, 1), \quad &\text{Second photo: } (1, 1, 1), \quad \text{Third photo: } (28, 28, 28) \\
\end{align*}

Looking at the pixels, you can guess that $(28, 28, 28)$ is an outlier and that $(1, 1, 1)$ is the actual background since these latter values appear the majority of the time. In order to “ghost” out the person, we would throw out the $(28, 28, 28)$ and use $(1, 1, 1)$ in our final image. So how can we algorithmically distinguish outliers from background pixels?

**Color distance**

To solve this problem, it’s handy to introduce the idea of “color distance” between pixels. A distance metric can help us to quantify how similar or dissimilar two colors are. This may initially seem like a daunting task, until we recall that a color is comprised of a red value, a green value, and a blue value. We can think of each pixel's color as a point in a 3-dimensional color space, with red-green-blue coordinates instead of the regular x-y-z coordinates.

Thinking in 2D: if we’re trying to find the distance between two points on a graph, we can use the distance formula (derived from the Pythagorean Theorem) to find the distance between them:

\[
distance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}
\]

We can apply this same principle to our 3D color space in order to define a distance metric!

\[
color\ distance = \sqrt{(R_1 - R_2)^2 + (G_1 - G_2)^2 + (B_1 - B_2)^2}
\]

Or in Python:

```python
math.sqrt(red_difference ** 2 + green_difference ** 2 + blue_difference ** 2)
```

**Note:** In Python “**” is a new operator that allows us to do exponentiation. Using this operator, $x \text{ ** } 2 == x \ast x$.

As a surprising optimization, we can actually omit the `math.sqrt()` for this algorithm (this will be explained further below).

---

Based on handouts and problems by Nick Parlante and John Nicholson (nifty.stanford.edu).
Ghost algorithm

Let's revisit our problem: for any particular \((x, y)\), look at the pixels across all images at \((x, y)\). We want to pick out the best pixel to use while avoiding the outlier pixels. The algorithm itself consists of two main parts:

1. **Compute the average RGB values**
   For a given \((x, y)\) pixel across all the images, average all the red values to get the average red value, average all the green values to get the average green value, and do the same for blue.

   For example, if a pixel at \((0, 5)\) had the following RGB values:
   - Picture 1: \((1, 1, 2)\)
   - Picture 2: \((1, 1, 1)\)
   - Picture 3: \((28, 27, 29)\)

   the averages would be as follows (using integer division):
   - The average red value would be 10: \((1 + 1 + 28)\) divided by 3
   - The average green value would be 9: \((1 + 1 + 27)\) divided by 3
   - The average blue would be 10: \((2 + 1 + 29)\) divided by 3

2. **Select the best pixel**
   To choose the best among the pixels, select the pixel with the smallest distance to the average red, green, and blue values. Equivalently, we could say “the pixel closest to the average RGB.”

   To think through the algorithm, imagine the pixels scattered in a 2D space of colors.

   ![2D mapping of true color, average of colors, and outlier values.](image)

   **Figure 3:** 2D mapping of true color, average of colors, and outlier values.

   All the pixels but the outlier will naturally cluster, grouped around the perfect color for that \((x, y)\) but displaced by little measurement errors. The outlier pixel will be off by itself, a completely different color. The average will fall somewhere in between but nearer to the cluster, since the cluster has many pixels and the outlier is just one pixel.

Based on handouts and problems by Nick Parlante and John Nicholson (nifty.stanford.edu).
As a result, selecting the pixel closest to the average will give you a pixel in the cluster of true color pixels.

**Note:** In order to select the closest pixel, we can omit the `math.sqrt()` in our color distance calculations. The pixel we choose has the smallest distance from the average pixel, and but due to the nature of squaring numbers, the pixel will also have the smallest squared color distance from the average pixel. In other words, comparing distance versus squared distance does not change the pixel we end up choosing. Therefore, we can leave out `math.sqrt()` and use distance squared instead of distance to find the best pixel (this is actually a common technique when dealing with large amounts of data!).

**Ghost code**

To complete the tasks described above, you should implement the following four functions in the order in which they are provided:

**Note:** Because we are using Pixel objects, which can be difficult to create and test with, we have written the doctests for you for these functions. Our tests are not comprehensive but should act as a sanity check.

1. **Task: Calculating squared pixel distance**  
   **Function:** `get_pixel_dist(pixel, red, green, blue)`

   Write the code for the `get_pixel_dist()` function, which returns the squared-distance between a pixel and a given red, green, and blue value.

2. **Task: Get average RGB**  
   **Function:** `get_average(pixels)`

   Write the code for the `get_average()` function, which takes in a list of Pixel objects and returns a list of exactly three elements: the average red value, the average green value, and the average blue value (each as an integer). In order for the doctests to work, you must return the values in the order [red, green, blue].

3. **Task: Find the best pixel**  
   **Function:** `get_best_pixel(pixels)`

   Write the code for the `get_best_pixel()` function, which given a list of Pixel objects, returns the best Pixel from that list according to the Ghost algorithm. As a reminder, the “best” pixel is defined as the pixel that is closest to the “average” pixel calculated by the function above. For example, when the `get_best_pixel()` function is called on the list `[(1, 1, 2), (1, 1, 1), (28, 27, 29)]`, it should return a Pixel with red/green/blue values of (1, 1, 2).

   If multiple pixels qualify as the best, `get_best_pixel()` can return any of the closest pixels.

---

Based on handouts and problems by Nick Parlante and John Nicholson (nifty.stanford.edu).
4. **Task: Create a “Ghost” image!**
   **Function: `solve(images)`**

   This is a high level function that solves the whole problem by calling other functions that solve the sub-tasks defined above. The provided `main()`, which you should not modify, does the housekeeping of loading the image objects, and then calls `solve()` to do the real work. You can assume that all of the images will be the same size. The code provided in `solve()` creates a blank image that is the same size as all the images in the provided list. It is your job to appropriately populate this image to create the “ghost” effect by writing the rest of the `solve()` function.

   For a helpful reminder about the SimpleImage functions you have available to you, check out the [Image Reference Guide](#).

**Decomposition**

If you decide to define helper functions while completing the above three functions, you should write function header comments explaining what these helper functions do. However, doctests are not required.

**How to run your code**

The `ghost.py` program takes just one command line argument – the name of the folder that contains the `.jpg` files. The folder "hoover" is relatively small, containing three `.jpg` files (the three example images at the top of this page). Run the program on the "hoover" files from the Terminal like this:

```bash
$ python3 ghost.py hoover
hoover/200-500.jpg
hoover/158-500.jpg
hoover/156-500.jpg
(solution image appears)
```

**Bigger test cases**

We have other places on campus where you can apply the ghost program! Each set of pictures is in its own folder: `clock-tower/`, `math-corner/`, and `monster/`. Look at the individual images inside each folder to get a feel for the data. Your code should work for all of these places (like test cases!). To try the different places, you can run the following commands in a Terminal.

**Clock Tower – a medium-sized case (Takes 5-7 seconds to run)**

```bash
$ python3 ghost.py clock-tower
```

**Math Corner – a large-sized case (Takes 8-15 seconds to run)**

```bash
$ python3 ghost.py math-corner
```
Monster – an extra-large-sized case (Takes up to 1-2 minutes to run)

$ python3 ghost.py monster

The monster images are each 1000 x 750; there are 8 image files; and each pixel has 3 numbers. How many numbers is that?

\[1000 \times 750 \times 8 \times 3 = 18000000\]

So in total, your code is sifting through 18 million numbers to figure out the ghost image!

For fun: You can even take your own pictures to test out your ghost code! You will want to keep your phone/camera as stable as possible (ideally using a tripod), since the algorithm does not work well if the background shifts.

Files to submit: ghost.py

Part 3: Parse Mystery (parse-mystery.py)

For this problem, you have received a text file that is hiding a valuable piece of information, and it’s your job to figure out what it’s hiding!

Parsing is a powerful tool for cleaning up and organizing large amounts of data. In this problem, you will be parsing large amounts of data in order to make sense of lines in a text file that look like this:

```
500 333
2X3b1b 832 ^248^n~ ~2(31 832 C^248^i P2P!3Yi1: 832 o^248^F~2i3z)1$ 832 ^248^MeW
2Hg3q1Cqs 832 b^248^ #F.2!B3$1Z@: 832 ,^248^9
```

While this might look like a bunch of garbled characters, this text is actually hiding some important numbers! Better yet, those numbers correspond to the RGB values of a secret image that you need to decipher. However, it looks like the RGB values have been badly corrupted. Each color channel (R, G, and B) has been corrupted differently, but all values for a given color channel seem to have gone through the same transformation. Luckily for us, the corrupted numbers are separated by spaces (this is also called “space-delimited”), meaning that the string ‘c:*23iy2U#% 632 C8^245^0r’ contains three values that represent one pixel’s RGB numbers: red hidden in ‘c:*23iy2U#’, green hidden in ‘632’, and blue hidden in ‘C8^245^0r’.

The file begins with a single line containing two numbers that specify the dimensions of the image. The first number is the width of the image, and the second number is the height of the image. Each of the lines afterward contains a random number of corrupted pixels. However, the red, green, and blue values associated with a single pixel will always be on the same line, and the values will always come in the order red, green, blue.
In order to see what’s hidden in the mystery image, you will need to write a handful of functions. For each of the following functions, except for parse_file() and solve_mystery(), you will need to write three additional doctests.

1. Task: Decorrupting red pixel components
   Function: decorrupt_red(s)

   This function takes in a string containing only the corrupted red value for a given pixel. In order to decorrupt the red values, you need to:
   - Remove the non-numeric characters from the string
   - Return the resulting value as an integer

2. Task: Decorrupting green pixel components
   Function: decorrupt_green(s)

   This function takes in a string containing only the corrupted green value for a given pixel. In the corrupted green values, sometimes the character ‘1’ has been turned into the character ‘a’. The number has also been flipped. To decorrupt the green values, you need to:
   - Turn any character ‘a’ into the character ‘1’
   - Reverse the string
   - Return the resulting value as an integer

3. Task: Decorrupting blue pixel components
   Function: decorrupt_blue(s)

   This function takes in a string containing only the corrupted blue value for a given pixel. In the corrupted blue values, the true blue value is hidden between the ‘^’ characters. There will only be two ‘^’ characters inside each corrupted blue value, but there may be 0 or more random characters preceding the first ‘^’ and/or following the last ‘^’ character. In order to decorrupt the blue values, you need to:
   - Retrieve the string between the two ‘^’ characters
   - Return it as an integer

4. Task: Parsing a single line
   Function: parse_line(line)

   In this function, you get to use the power of the “decorrupting” functions that you just wrote! This function takes in a string containing a random number of corrupted pixels (represented by space-delimited RGB values) and returns a list of integers corresponding to the decorrupted values. The list that you return should be in the same order that the corrupted numbers appear in the string. You can make the following assumptions about the string line passed into your function:
   - The line only contains complete pixels. In other words, if a given pixel is contained inside the string line, all of its RGB values will be in the line.
   - The RGB values for each pixel will always appear in the correct order: red, green, and then blue.

Based on handouts and problems by Nick Parlante and John Nicholson (nifty.stanford.edu).
• The above bullet points also mean that the number of valid space-delineated numbers in a given line will always be a multiple of 3.

5. Task: Parsing an entire file
   Function: parse_file(filename)

Now it's time to read the entire file and parse all of its hidden numbers! This parse_file() function should make use of your parse_line() function and should return a list containing the width of the hidden image, the height of the hidden image, and all of the red, green, and blue integer values hidden in the corrupted text file.

Remember that the first line of the file contains two integers, which represent the width and the height of the image. When processing the file, these two numbers should be the first two elements of your outputted list.

We don’t expect you to write doctests for this function, but you should still test your code as you go. To help you with testing, we’ve provided test code in main() that will help you examine the output of your function by printing the list it returns. Note that the two small test files we provide, texts/3lines.txt and texts/10lines.txt, do not actually correspond to a hidden image. They exist only for your to test your parse_file() function. You should take a look at these files directly to make sure your outputted list looks reasonable.

To run the code that tests only parse_file(), you should run the parse-mystery.py program with the command line argument -nums, followed by the file you want to test. For example, to test on the sample file of only 3 lines (texts/3lines.txt), you would run the following in your Terminal:

$ python3 parse-mystery.py -nums texts/3lines.txt

The above line should result in an output that looks something like this:

[5, 5, 245, 8, 243, 186, 251, 179, 226, ...]

Since this is just a test file, the above output does not correspond to an actual image, but if it did, you would expect the first 5 to represent the image's width, the second 5 to represent the image's height, the first pixel's RGB representation to be (245, 8, 243), the second pixel's RGB representation to be (186, 251, 179), etc.

6. Task: Solving the mystery
   Function: solve_mystery(filename)

Finally, we’re going to solve the mystery of the hidden picture by putting all of your functions together and using them to generate a new image! Your solve_mystery() function should do the following:

Based on handouts and problems by Nick Parlante and John Nicholson (nifty.stanford.edu).
- 14 -

- Parse all the numbers from the given filename
- Create a blank image of the correct width and height
- Loop over the image to fill in its RGB values from the parsed file

We’ve provided starter code for the `solve_mystery()` function, but you should fill in all the parts indicated by the comment "### YOUR CODE HERE ###."

**Hint:** Think about which values you want the first time the loop runs. What about the second time? The third time? Drawing this out on paper can help you work out a pattern.

The provided `main()` will call your `solve_mystery()` function when you give the program exactly 1 command line argument with the file you want to parse. For example, the following command runs your program on `texts/mystery_1.txt`:

```
$ python3 parse-mystery.py texts/mystery_1.txt
```

There are two additional mystery files that you should discover the contents of: `texts/mystery_2.txt` and `texts/mystery_3.txt`. Your code should take about 5-15 seconds to run and should be able to solve all of these different text files (but not `3lines.txt` or `10lines.txt`, as these do not contain images).

Files to submit: `parse-mystery.py`

Congratulations! You’ve implemented your own bluescreen algorithm, made people disappear from images, and cracked the case of the parse mystery!

---

Based on handouts and problems by Nick Parlante and John Nicholson (nifty.stanford.edu).