Handout 8 - Assignment 3

Written by Mehran Sahami

April 15, 2022

Assignment #3: Lists and Images

Due: 12:15pm (Pacific Time) on Monday, April 25th

Based on problems by Nick Bowman, Sonja Johnson-Yu, Kylie Jue, Nick Parlante, Eric Roberts, and the current CS106A staff.

This assignment consists of two sets of programs to give you practice with lists and images, respectively, in Python. You can download the starter code for this project under the "Assignments" tab on the CS106A website. The starter project will provide Python files for you to write your programs in.

The assignment is broken up into two parts. The first part of the assignment focuses on writing a function and a program using lists, which you can get started on now. In the second part of the assignment, you'll get the chance to do some image manipulation, including a sandcastle (warm-up) problem as well as two longer programs to write. We'll cover images in class on Monday, April 18th, so you can do the second part of the assignment after that.

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 and also considering their ethical implications!

Ethics in Computing

As we mentioned at the beginning of the quarter, being a computer scientist entails more than just writing code. The Association for Computing Machinery's Code of Ethics (https://www.acm.org/code-of-ethics) begins, "Computing professionals' actions change the world. To act responsibly, they should reflect upon the wider impacts of their work, consistently supporting the public good." Starting with this assignment (the portion on image manipulation), you will put your ethical thinking skills to use by reflecting on the potential impacts of your work and how they align with your values. The programming fundamentals that you are mastering give you the ability to create powerful, expressive, and impressive applications – just remember that with great power comes great responsibility!

Part 1: Lists

1. **"Sandcastle" (warm-up) problem** You should write a function called **greater_than** (in the file **greater_than.py**) that is passed a threshold integer value and a list of integers, and returns a *new* list which contains only the numbers *strictly greater* than the threshold value from the original list passed in. For example, if your function was called as follows:

greater_than(6, [20, 6, 12, -3, 14])

then it should return the new list:

[20, 12, 14]

Note that the 6 in the original list is **not** included in the result list since it is not strictly greater than the threshold value, which is 6.

If your function is called with the empty list (or a list that does not contain any values greater than the threshold value), as follows:

greater_than(0, [])

then it should return a new empty list:

[]

Doctests are provided for you to test your function. Feel free to write additional doctests if you would like practice with that aspect of Python. Tests for your function are also provided in the **main** function included in the program.

2. **Reading values into a list and removing duplicates**

In the file **removeduplicates.py** you should implement two functions. The first, called **read_list()**, asks the user for a series of integers until the user enters 0 to stop. The integers entered by the user (not including the final 0) should be stored in a list that the function should then *return*. Here is a sample run of the user entering values (user's inputs are in ***italics***) and what the function should return (internally) in the program:

Enter value (0 to stop): *5*

Enter value (0 to stop): *3*

Enter value (0 to stop): *6*

Enter value (0 to stop): *2*

Enter value (0 to stop): *7*

Enter value (0 to stop): *6*

Enter value (0 to stop): *3*

Enter value (0 to stop): *3*

Enter value (0 to stop): *0*

If the user entered the values above, the function should return the list:

[5, 3, 6, 2, 7, 6, 3, 3]

The second function you should write (also in the file **removeduplicates.py**) is called **remove_duplicates(num_list)**. This function is passed a list of integers (**num_list**) and it should create and return a *new* list which does not include any duplicate values from the original list passed in. The original list passed into the function (**num_list**) should not be changed. For example, calling:

remove_duplicates([5, 3, 6, 2, 7, 6, 3, 3])

should return the following new list:

[5, 3, 6, 2, 7]

If your function is called with the empty list, as follows:

remove_duplicates([])

then it should return a new empty list:

[]

Doctests are provided for you to test your **remove_duplicates** function. Feel free to write additional doctests if you would like practice with that aspect of Python.

Part 2: Images

Setup: Installing Pillow

Before you get started on Part 2 of the assignment, make sure that you have run through the Pillow installation instructions, which can be found in the "Image Reference" handout on the CS106A website. If you cannot get Pillow installed successfully, please come to LaIR or post on Ed for help.

Ethics Questions

After completing certain components of Part 2 of this assignment, we will ask you to reflect on the ethical issues raised by the code you have just written by answering a couple of free-response questions. In most cases, the ethical context for the questions will have been touched upon during class (for images, that will be in the lecture on April 20th), but sometimes we may present a brief description of new contexts as well. We do not believe that there is one right

answer to these questions. All we ask is that you think critically and *answer the question to the best of your ability in a short paragraph (2-4 sentences)* in the text file **ethics.txt** provided in the Assignment 3 folder.

Reasoning about these questions will help you to understand and evaluate the impact of the computer programs you write and instill you with skills that you'll use throughout the rest of your journey as a computer scientist. We encourage you to discuss these questions and the issues they raise with your section leaders at IGs, with your fellow students, and with the Embedded EthiCS Fellow, Katie Creel.

Part 2 To Do

1. **"Sandcastle" (warm-up) problem: Finding forest fires.**

   We're going to start by writing a function called **highlight_fires** (in the file **forestfile.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 some intensity threshold. In this case, we have provided you with an appropriate intensity threshold of 1.05 via a constant named **INTENSITY_THRESHOLD** in the file **forestfire.py**. Note that this is a different intensity threshold value than we used in class for the "redscreening" example, as different applications often require different intensity threshold.

   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 Figure 1. On the left side of Figure 1, we should the original image for comparison.

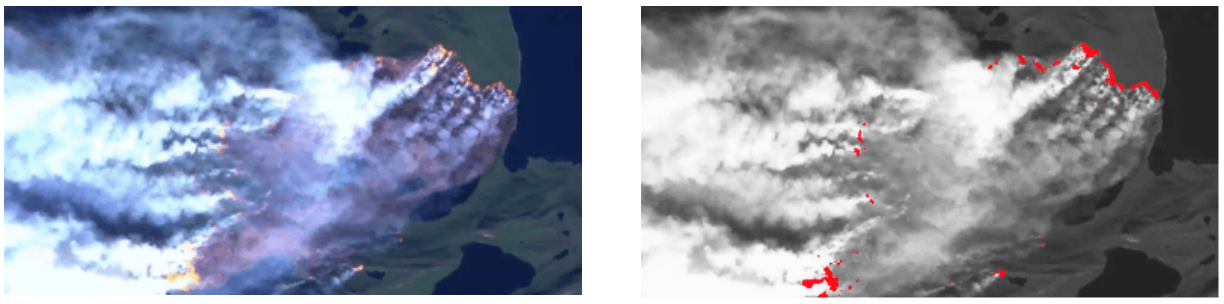


Figure 1: Original forest fire image on left, and highlighted version of image on right.

   For a helpful reminder about the SimpleImage functions you have available, you can use the Image Reference handout (available on the CS106A webpage).

   Ethics Question

   In this exercise, you've used the power of computing to accomplish a real-world task: recoloring satellite imagery to help humans understand it. Your tool could make it easier for a human to identify which areas of the forest are on fire. This could help firefighters to plan a fire containment strategy or to decide which homes need to be evacuated. You've used image manipulation to highlight and accentuate an existing phenomenon, making it easier to comprehend via the naked eye. In the process, you have reduced or eliminated other color information in the image. This kind of visual idealization is common in science. Astronomers use coloration schemes to display different types of non-visual measurements and biologists use image processing tools to accentuate potentially diseased areas of tissue.

   However, idealizations can also be dangerous. Their use relies heavily on the assumptions built into their creation and the match between the idealization and its purpose. Thinking back to the program you've written and looking more closely at the output image, you can see that the red highlighting of the fires is not perfect. Some areas that are on fire are not highlighted and other areas that are highlighted are not actually on fire. The consequences of this on a firefighter's plan of attack for the fires or an evacuation administrator's

evacuation plan could be dire – lives and homes could potentially be lost. Thus, we want you to consider and answer the following questions (write your answer in the file **ethics.txt**):

a. What is the programmer's responsibility in knowing the context of evaluation of an image processing algorithm/tool? How should this affect their decision-making process when implementing the algorithm/tool?

b. What information should you convey with your image in order to allow the viewer to evaluate its limitations? How would this change with the context of use?

2. **Warhol Images**

Your next task is to write a program that generates Andy Warhol-style images that includes repeated variants of a starting image. The name of the image file that you will be using for the starting image is specified in the constant **IMAGE_FILE**, which is provided for you. To tackle this task, we're going to break it down into a two primary functions (although you are certainly welcome and encouraged to write additional functions to decompose your program). You should implement the following two functions in the order in which they are listed below. Empty versions of these functions are provided in the file **warhol.py** for you to fill in. As mentioned, you can add additional functions to the **warhol.py** file as needed, but the functions specified below are meant to be milestones in developing your program.

a. **Task: Create a filter for a single image**

Function: **create_filtered_image(red_scale, green_scale, blue_scale)**

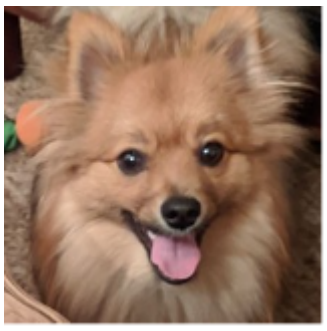
The first step is to create a function that can create filtered versions of the starting image. The function is passed in floats (real values) for **red_scale**, **green_scale**, and **blue_scale**. The function should return an image object that is a copy of the image in the file specified by the constant **IMAGE_FILE**, where the red, green, and blue values for each pixel are respectively scaled by the parameters **red_scale**, **green_scale**, and **blue_scale**. In order words, for **every** pixel in the original image in the file **IMAGE_FILE**, the corresponding pixel in the resulting filtered image should have:

new red value = old red value * red_scale

new green value = old green value * green_scale

new blue value = old blue value * blue_scale

In the starter file we give you, the constant **IMAGE_FILE** is set to **'images/simba.jpg'**, which is the name of an image file of Simba the Dog:



If you were to call your function as follows:

create_filtered_image(1.5, 0.7, 1.5)

then your function should return a new image object that would look as shown below if it was displayed:

Note that this function does **not** save an image file to disk and does **not** modify the original file containing the image. This function reads the image from the file into an *image object* (in memory when the Python program is running) and all color changes are made to that image object (in memory).

b. **Task: Create a Warhol-style image**

Function: **make_warhol()**

You should write a function **make_warhol** that returns an image object, which has the image in the file specified by the constant **IMAGE_FILE**, copied 6 times (in 2 rows and 3 columns), where each patch (subimage) gets recolored using your **create_filtered_image** function. An example resulting image is shown below:

You should select different **red_scale**, **green_scale**, and **blue_scale** values for each of the patches, so they look distinct. You don't need to try to match the colors in this example exactly. The picture here is just an example. Experiment with different combinations of **red_scale**, **green_scale**, and **blue_scale** values. For example, the pink Simba in the first (upper left) patch in the image above was generated by calling:

create_filtered_image(1.5, 0, 1.5)

The choice of the actual **red_scale**, **green_scale**, and **blue_scale** values for each patch in your program are left up to you. Be creative! This sort of image is inspired by some of Andy Warhol's paintings, which is where the name for this function comes from.

As before, this function does **not** save an image file to disk and does **not** modify the original file containing the image. This function reads the image from the file into an *image object* (in memory when the Python program is running) and generates a new image object (in memory) that contains the different colored patches.

Hint: Your **make_warhol** function will need to start by creating a new blank image object (using the function **SimpleImage.blank**) in order to then copy individual patches into. In order to do that, you will first need to determine the width and height of the original image so you can determine how large the new blank image you are going to create should be.

Writing the program

There is a **main** function provided in the starter code which simply calls your **make_warhol** function and displays the resulting image returned from your function.

To help you write and debug the function **create_filtered_image** *before* you write your **make_warhol** function, the starter code also contains a call to the **create_filtered_image** function, and displays the image returned from that function. The code that calls **create_filtered_image** and displays the resulting image is just provided for debugging purposes so you can complete the first task in this problem (creating a single image filter) before moving to create full Warhol images. You can delete the code in the **main** function

that calls **create_filtered_image** and displays the results when you think you have the filter for single images working, and then just make use of the call to the **make_warhol** function to show the final result of your program.

3. **Ghost**

In the final problem, we're going to use multiple images to create a new image (with some "clutter" removed). Suppose we are trying to take pictures of Stanford, but each image has people walking through the scene, as shown in Figure 2 below:

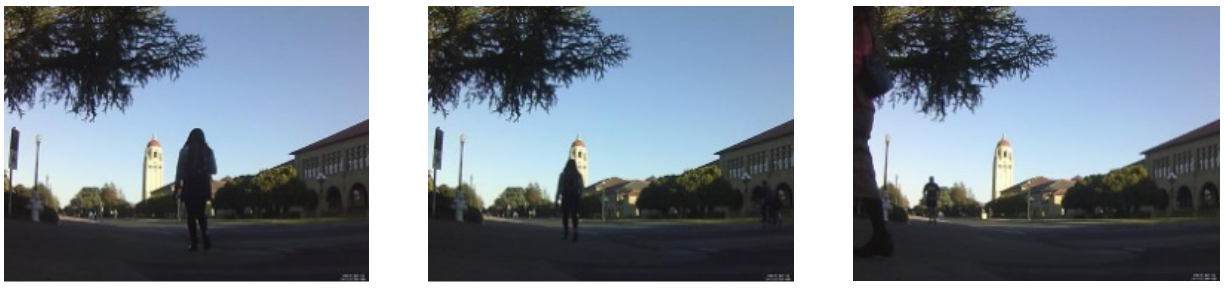


Figure 2: Three images of Hoover Tower, with pedestrians walking in the images.

We'd like to analyze all these three images and figure out a way to "ghost" out all the people and make them disappear, giving us a clear view of Hoover tower, as shown in Figure 3. For this part of the assignment, you will be writing a program called **ghost.py** that implements this functionality. An algorithm for solving this problem is explained below. **Please also see the ethics question for this part of the assignment below**.

Figure 3: A sample output of your program in **ghost.py**.

Ghost algorithm motivation

Let's say that we have three 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 because there's a person at that location in that image. We'll call pixels where there is a person "outlier" pixels, because they are different from all the other pixels that exist at that that same **(x, y)** location in the other photos. We'll assume that outlier pixels are always in the minority – that is, there will always be more "regular" pixels than outliers.

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

First photo: (1, 1, 1), Second photo: (1, 1, 1), Third photo: (28, 28, 28)

Looking at the pixels, you can guess that (28, 28, 28) is an outlier and that (1, 1, 1) is the actual background since the RGB values (1, 1, 1) appear the majority of the time among the pixels at location x=100, y=50. In order to "ghost" out the person, we would throw out the (28, 28, 28) and use (1, 1, 1) as the RGB values for the pixel at location x=100, y=50 in our final image. So, we need a way to algorithmically distinguish outliers from background pixels. We explain that below.

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!

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

In Python, we could write that distance function as:

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

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

Ghost algorithm

Let's revisit our problem: for any particular **(x, y)** location in an image, look at the pixels across all images at that same **(x, y)** location. We want to pick out the "best" pixel to use while avoiding the outlier pixels. The algorithm for doing this consists of two main parts:

a) **Compute the average RGB values**

For a given **(x, y)** location 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, consider a pixel at location (0, 5), which had the following respective RGB values across three images:

- Picture 1: (1, 1, 2)
- Picture 2: (1, 1, 1)
- Picture 3: (28, 27, 29)

The average value for each color across the pixels 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

b) **Select the best pixel**

To choose the best among the pixels, select the pixel with the ***smallest*** color 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 more generally, imagine the pixels scattered in a 2D space of colors.

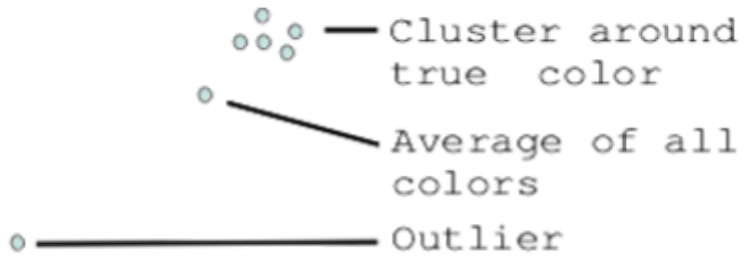


Figure 4: 2D mapping of true color, average of colors, and outlier values.

All the pixels but the outlier will naturally cluster together, 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.

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 three functions in the order in which they are listed below. Empty versions of these functions are provided in the file **ghost.py** for you to fill in. Of course, you can add additional functions to the **ghost.py** file as needed, but the functions specified below are meant to be milestones in developing your program.

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 provide some simple checks. You are certainly welcome (and encouraged) to write your own doctests as well. This will give you a chance to get practice with Python's very useful doctest feature.

a) **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.

b) **Task: Find the best pixel**

Function: **get_best_pixel(pixel_list)**

Write the code for the **get_best_pixel** function, which is given a list of Pixel objects and 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. For example, if the **get_best_pixel** function is called on a list containing three Pixels with the following RBG values: (1, 1, 2), (1, 1, 1), and (28, 27, 29), it should return the Pixel which has RGB values (1, 1, 2).

If multiple pixels qualify as the best, **get_best_pixel** can return any of the closest pixels. You can assume **pixel_list** will never be empty.

c) **Task: Create a ghost image**

Function: **create_ghost(image_list)**

This is the function that solves the whole problem and should make use of the other functions you wrote to solve sub-tasks. The **main** function we provide for you, which you should not modify**,** does the housekeeping of loading the image objects, and then calls **create_ghost** to do the real work. It will then display the image returned by your **create_ghost** function. You can assume that all the images in the list of images passed into this function are the same size (have the same height and width) and that the list of images is never empty. Remember that your function should create a new (blank) image of the same size as the images passed into the function and the appropriately set the pixels in this new image to create the "ghost" effect.

Ghost Decomposition

If you decide to define helper functions while completing the above 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 one "command line" argument, which is the name of the folder that contains the image files you would like to run the Ghost algorithm on. A command line argument is just something that you type on the terminal command line after the name of the program you want to run. For example, to run the **ghost.py** program using the images in the "hoover" directory, you would type:

On the PC: **py ghost.py hoover**

On the Mac: **python3 ghost.py hoover**

Running the program on the "hoover" files from the Terminal like this will print out the names of the images from that directory that your program is being run on. It should (after you write your solution), then display the image returned by your **create_ghost** function. For example, the output on the terminal would like:

> py ghost.py hoover

Loading hoover\156-500.jpg

Loading hoover\158-500.jpg

Loading hoover\200-500.jpg

Displaying image!

(then solution image appears)

Additional test cases

In addition to the **hoover** directory, we have provided image of other places on the Stanford where you can apply the ghost program! Each set of pictures is in its own folder, which are:

clock-tower
math-corner
monster

Look at the individual images inside each folder to get a feel for the data. Your code should work for all of these image folders. To try the different places, you can run the following commands in a Terminal (use "python3" instead of "py" if you are on a Mac).

Clock Tower – a medium-sized case (Takes a few seconds to run)

py ghost.py clock-tower

Math Corner – a large-sized case (Takes a few seconds to run)

py ghost.py math-corner

Monster – an extra-large-sized case (Can take a few minutes to run, so be patient)

py ghost.py monster

Note that in some of the resulting images, you might see some "ghost-like" artifacts of people/bicycles that have been removed from the original images. This is normal (though you might want to think about why it happens).

Ethics Question

In this part of the assignment, you've implemented a clever algorithm that eliminates transient passings of people in many partial images to build a single image of Stanford's campus as empty. Your Ghost algorithm performs an "idealization," removing components that are deemed to be impediments to the presentation of the desired version of the image. Taking this into consideration, we want you to consider and answer the following questions (write your answer in the file **ethics.txt**):

a. For what purposes might the final image produced by your Ghost algorithm, or other empty images of other parts of the world, be used? Who might want to create such an idealization?

b. In class (on April 20th), we defined manipulation as hidden influence that subverts another's decision-making powers. Of the purposes you listed in (a), which ones involve *manipulation* and why?