Introduction

Ever wondered how Instagram filters work? Or how Photo-shop is able to finely tune out the areas you want to overlook and zoom in on the ones you want to highlight? This project allows you to go behind the scene and explore image manipulation directly, from learning how to blur an image to combining two distinct photos! By the end of the project you will have created a collage of unique images that you can showcase (all done through code)!

Helpful terminology:

Our images will have 3 values associated with them: an x-pixel value, a y-pixel value, and then a number of channels. The first two simply refer to the size of our image, while the last one tells us what type of image we're using (RGB, black and white, etc). To edit these images we will be storing these values in an array, where you can get a specific pixel by specifying Image.array[x-pixel][y-pixel][channel].

A numpy array is similar to a list, except it is much faster to use. It takes up less memory, and also has a few of its own extra functions that you can use on it. With numpy arrays you can make arrays that are more than 1-dimension! The one thing to note about numpy arrays are that they always have to have the same data type in them (if at index 0 you have an array then the numpy array must contain an array at index 1, 2, 3, etc).

> *Stop and think*: Why we would use a numpy array instead of a regular list for this project?

We will also be using a class called `Images` for our implementation. Because of its complexity we will be giving it to you, and as a result you will not have to focus on coding it! You can download it from the canvas site, and we will explain how it works in the next section.

> *Stop and think*: Taking into account what we know about an Image object and how it is defined, what are some functions that you think the Image class has?

Installing and Importing Modules

Before we even look at the `Image` class, let's quickly install a few things we're going to need! You might have noticed that at the very top of the file we have something that says ïmport numpy". This will allow us to use any methods in numpy and numpy arrays! The only problem is that VS code doesn't seem to recognize numpy as a module to import. That's because we haven't installed numpy yet!

To install Numpy all we have to do is type **python3 -m pip install numpy** in the terminal. Installing might take a while, but you won't have to install numpy again after this! Once numpy is installed, we want to make sure that our code has access to it. Run **pip3 -show numpy** in the terminal, and look for where it prints location. There should be a path link that you can hover over. Open the folder in a new window, and you should see a bunch of files. We only care about the one titled numpy though!

Hover over the numpy file and right click it. You should see something that says 'Reveal in File Explorer'. Click on that, and you will see numpys physical file location. Drag the numpy folder to wherever your Image Manipulation folder is, and place it inside the folder. Yay, you can now use numpy! This process will be repeated whenever we mention installing/importing a module.

The `Image` Class

This image class is the foundation of our code, and is responsible for transforming the image so that we are able to directly manipulate it. It has the ability to define/create an image object, read an image file (convert it to pixels), and write an image file (convert our code back to a png).

The `__init__` Function

This function initializes the image object, taking in the dimensions and the channel of a png file or a file name. If the former, the function maps each of the inputs to a self variable that it uses to create an array. If you pass in a file name though the function will be able to directly read and convert the file using the `read_image` function (more on this in a bit!). It then reassigns all of the values (x-pixels, y-pixels, number of channels) by using `self.array.shape`.

The `read_image` Function

`read_image` is able to take in a file name and convert it to a numpy array using the `Reader` function from the `png` class to do so. We then resize the array!

The `write_image` Function

This function converts our code back to an actual image! We initially clip (or restrict values) our array from 0-1 and then later re-scale it. It finally uploads our new modified image to the output directory.

> *Stop and think*: Why might we want clip the image's values from 0-1? (Hint: Remember that we will be re-scaling the image, and that RGB values, for example, have specific values that they can go up to. What is that range and would we want to go outside of it?)

Let's Begin!

Now we're ready to start creating our first photo! All of the functions to do are in **Image_Manipulation.py**
We will be coding two functions: the `brighten` function and the `adjust_contrast` function. Once you have implemented both there is also a challenge that needs to be done before you ask for a checkoff. The challenge will use similar methods to the functions that you coded earlier!

Coding the `brighten` Function

Our brighten function will light up any area that the user wants to by a certain amount (which we will be referring to as a 'factor'). It will be taking in three parameters: the Image, a factor, and pixels_to_brighten. For now we're going to ignore the last one and focus on the first two parameters.
Our first step will be to create a new Image that is exactly the same as the one that we passed in. This might seem redundant, but it's actually a very important step!

> *Stop and think*: Why do you think we need to create a new Image in the function? These Images will be exactly the same, so what do you think the point is? (Hint: Remember that you might want to use the same image for multiple manipulations/functions. What would happen if you used the same image for all of them?)

Once we have our new Image, we have to re-get all of its values. Our program currently knows and stores nothing about this Image, and as a result we can't directly modify it until we get this information.

Now it's your turn:
Create a new Image Object and store its x-values, y-values, and number of channels in their respective variables. (Hint: you can directly access these values using a method found in the Image class)

Brightening the Entire Image:

We now have code that creates an Image Object and stores its properties! We haven't changed anything about the image yet though, so it's still exactly the same as the one we initially passed in. To make the picture brighter, we need to go through every aspect of the image and brighten it by the factor. But before we do that, let's think about an Image's setup.

We have rows and columns of pixels, each with it's own x coordinate and y coordinate (you can think about an Image like a giant board that is divided up if that's easier). Each one of these pixels also has its own color component that comes from the channels that make up it. For example, if we were working with an RGB Image, a pixel would have varying levels of Red, Green, and Blue to create the color we see on it.

(0,0) Pixel_1: Red, Green, Blue	(0,1) Pixel_2: Red, Green, Blue
(1,0) Pixel_3: Red, Green, Blue	(1,1) Pixel_4: Red, Green, Blue
(2,0) Pixel_5: Red, Green, Blue	(2,1) Pixel_6: Red, Green, Blue

We want to modify each of these pixels by the number of channels the Image has (as that is the amount of color we will be altering). If we look at and change each one of these elements, we will eventually brighten the entire image!

Now it's your turn:
Write the code that will be able to go through each pixel and their respective channels. Once you're at a specific index, multiply it by the factor! (Hint: This sounds like a lot of iterating and traversing! What would be an easy way to go through each of these elements? You might require multiple/nested versions of these). You can call a specific pixel of an Image by saying Image.array[x value][y value][channel value].

Stop and think: While this process works, it's a bit lengthy. Is there a faster way to code this, taking into account the fact that Image can be stored as an array?

Nice, we now have a new Image object! But we haven't done anything with it yet: for the user to actually see it we'll need to return it.

Now it's your turn:
Return the image outside of all the loops!

Great job! You should now have a function that takes in an image and factor, and either returns the image darker or lighter depending on the factor amount. Let's try to test this out! at the very end of your file, under `if__name__== '__main__'`: create an image object and try the function you just made!

Now it's your turn:
Create an image object that contains the filename/path to the image you want to modify. Then, call the brightened function on it, making sure to enter the factor based on whether you want to dim it or brighten. Finally, place your finished product in the output directory (check the Image class if you don't remember how to do this)! Make sure the image is originally a png file as the code will not work if you have to convert from a jpeg. You will do this on every picture you create so feel free to reference this!

Look at your image and see if you're satisfied. If you are, run the `Image_test_case` file and make sure that `brightened_image` and `dimmed_image` pass (Note: there is a specific way to run the test file so please read the Test cases section at the end of part 1 with detailed instructions. You can ignore the comments in the actual test case file, as I have already fixed the 'input' and 'output' error!). Here is an example image:

Abbildung 1: Original Image

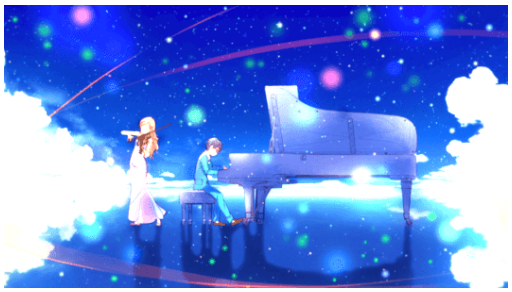


Abbildung 2: Brightened Image

Coding the `adjust_contrast` Function

Great, we now have one function done! The next one we want to focus on is the `adjust_contrast` function. What `adjust_contrast` does is it defines a midpoint value, and then changes the current pixel value based off of how much it differs from it.

Abbildung 3: (Credit: RaeD09 at English Wikipedia, CC BY-SA 3.0) [1]

Abbildung 4: High Contrast Image (Modified earlier image)[2]

The code is going to be fairly similar to how we did the `brighten_image` function, as once again we are going to have to iterate through each index of the image and get the original pixel value. This time though, we are going to try to find the difference between the pixel and a user inputted 'midpoint' value, and increase it by a factor. We then need to add the midpoint value back so that it contributes to the contrast! [1] [2]

> Now it's your turn:
> Copy your code from the `brighten` function, editing only the line where you edit the current pixel's values. Instead of multiplying the pixel by a factor, do the appropriate operations mentioned above.

Look at your image and see if you're satisfied. If you are, run the Image test case file and make sure that `high_contrast_image` and `low_contrast_image` pass.

[1] https://commons.wikimedia.org/wiki/File:A_photo_of_a_lake.png
[2] https://creativecommons.org/licenses/by-sa/3.0/deed.en

Challenge: Making a Sepia Filter

You've probably seen this filter before: it's one of the most common, and dates back to the late 1800s. The filter 'warms up' the image and can also be used to give it a more 'vintage' feel. Today, we will be replicating this filter with simple Image Manipulation!

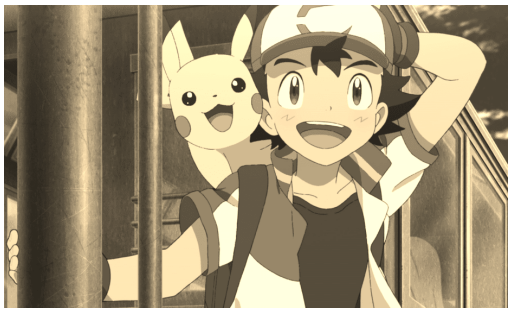


Abbildung 5: Ash and his trusty Pokemon, Pikachu

Abbildung 6: Ash and his trusty Pokemon Pikachu, (Colorized, 2022)

(Disclaimer: The above caption is a joke and the first picture has been modified. I do not own the second picture and have not colorized it. Please don't sue me Pokemon :))

The module that we've been using and importing so far is PyPNG, but there are other classes that we can import to mess around with images. One of the most common is Pillow which comes from PIL, the Python Imaging Library. To use this module though, the first thing we're going to do is install it. To install it, run **python3 -m pip install –upgrade pip** in the terminal. Once that's finished you can type and enter (in the terminal) **python3 -m pip install –upgrade Pillow**. If this installation is not working, feel free to come to Office Hours and ask for some help!

To make a sepia filter, you need to modify each channel of an RGB image. Since we're using Pillow, the process is going to be a bit different than before!

Our first step is to open our file, and then convert it to an RGB image. You can see all of the methods available in the Pillow module here: https://pillow.readthedocs.io/en/stable/reference/Image.html. Try to see if you can find the methods needed for both of these steps! (Hint: Once you have the Image open, all you have to do to convert to an RGB image is use the function **convert**.)

> Now it's your turn:
> Open the Image and then convert it to an RGB image.

Once that's done, we now need to loop through each pixel like before! There's one small difference though: this Image we now have does not stores the RGB values in channels: instead, once you have the x and y value of the pixel, you are able to directly access the RGB values as a list. They take the form of [R,G,B].

> Now it's your turn:
> Loop through all of the pixels. Remember that you don't need to go through the channels in pixel. (Note that we might have to use a different method to find out the **size** of the image (is this a hint?). Look through the docs and see if you can find it!).

Once we have access to a specific pixel, we need to do some re-calculating for each of the RGB values! (Note that these formulas come from the Microsoft, reported by the TechRepublic).

New Red = (Red*0.393)+(Green*0.769)+(Blue*0.189)
New Green = (Red*0.349)+(Green*0.686)+(Blue*0.168)
New Blue = (Red*0.272)+(Green*0.534)+(Blue*0.131)

You can get each of the original pixel values by using a specific method that returns a pixel (look for this in the documentation!). If any of these values turn out to be greater than 255, set the specific one equal to 255 (so if New Blue is 277, you would set it equal to 255).

> Now it's your turn:
> Calculate each of these new values using the formula above. Remember that the pixels have a list stored in them that has the RGB values represented as [R,G,B].

Finally, the last thing we want to do before saving the Image is putting the pixel back! In the image you converted to RGB, use the method **putpixel** to place the RGB values at the specific pixel you're at. Once that's done, use the method **save** to save the new Image to a specific directory!

> Now it's your turn:
> Put the pixels back into the Image and save it! Now look at the Image you made!

Running the Test Cases

Before you do your checkoff, I'm going to quickly explain how to run the tests! For the test cases to work you're going to need to import a class called Open-CV. To install it, all you have to do is type and enter **pip install opencv-python** in the terminal. This will allow you to easily compare images!

Don't delete images that are already in the **text_images** folder. They are there so you can see how the pictures you produce compare to the ones that are there for testing. You will need to create another folder called **input** (inside the Image Manipulation folder), and place the **text_images** inside of it. This would already be done but there was a problem with zipping the sub folders!

You might notice that the test cases take a while. Don't worry, that is because of the `blurry` function! If you want the test cases to go faster you can delete the code (in the `test_cases.py` file under `def test_blur`) and put pass. Make sure that you copy the code though so that you can put it back later, otherwise you will not be able to test the `blurry` function!

> **Checkoff - Part 1**:
> In Part 1 you should have:
>
> - Coded the `brighten` function
> - Made `adjust_contrast` function
>
> Show a course staff member your working code to get the checkoff.

In the next part we will:

- Create the `blur` function
- Code the `apply_kernel` function
- Add Motion Blur to a Picture

Continuing on our Image Manipulation Project, in this part we will:

- Create our `blur` and `apply_kernel` funnel
- Modify an Image to make it look like it has motion blur

Let's get right into it!

The `blur` function

Blur is extremely important in photography: not only does it help draw the viewer's attention to a specific focal point, it can also remove blemishes and even convey movement (which we'll get to in a bit)!

> *Stop and think*: What does the word 'blurry' mean to you? What do you think causes us to view objects/pictures as blurry?

Abbildung 7: Original Image

Abbildung 8: Blurred Image

How do we get these smeared images though? The easiest way is by taking a single pixel, and then changing its value to the average of all the pixels near it. You then repeat this for every pixel in the image, and eventually you will have a thoroughly blurred and interconnected image!

> *Stop and think*: According to this method of blurring an image, how can you increase the blurriness of an image? How can you make it less blurry?

Overview

The `blur` function takes in two parameters: an Image object and kernel_size, which is basically the amount of adjacent pixels we will be blurring with the image. For example, if kernel_size=11 and we are blurring the image in every direction, we would look at the 5 pixels on the left, the 5 on the right, the 5 above, the 5 below, and the 5 above and below on both diagonals.

> *Stop and think*: Do you think there are any constraints for the value that we can have for the kernel_size?

Coding the `blur` Function

Like every single time we want to modify an image, we need to create a new Image object. Once that's done though, we need to create a variable that will tell us how many neighbors a pixel has on each side. Remember that in the kernel_size=11 example we had 5 pixels that we wanted to examine on each side.

Now it's your turn:
Create a new Image Object and store its x-values, y-values, and number of channels in their respective variables. Then calculate and declare a neighboring_pixel variable, which should be equal to the amount of pixels that we will be using on a single side of a pixel. This will be important when we are trying to determine the current pixel value from the average of its neighbors!

Now that we have the basic set-up done we can actually start working on blurring the object!
Once again we will need to go through each x-pixel, y-pixel, and num-channels to access every single individual pixel. This is going to be the exact same code that we had in our `brighten` and `adjust_contrast` function, and we will be using it in almost every single function we create for this project.

Now it's your turn:
Iterate through each x-value,y-value, and num-channel to access every individual pixel object.

Once we arrive at a specific pixel we need to make a decision: how do we want to get the average of all of the nearby pixels?

We need to look at all the pixels that are to the left, to the right, above, and below the current pixel, and add them to a total as long as they are in the neighboring_pixel range. This total should be initialized as soon as you enter a specific pixel (so it should be made in one of the main for loops that we had from earlier!)

Now it's your turn:
Get the values from each of the neighboring pixels and add them to a variable titled 'total'. You will find an additional loop in the x and y direction extremely help for this. (Hint: the range for these two loops should be so that we are only looking at the neighbor pixels. Remember that we have a variable defined that lets us know how many pixels before and after that we have to look at!)

Stop and think: It's very easy to assume that we can just check every single pixel in the range nearby, but we have to remember that pixels can be on the side, which means that we might not always have pixels on our left/right/above/down (ex: if we are at index (0,0) then there will not be any pixels to the left or above). How can we prevent our code from checking non-existent pixel values?

Now that we've looked at all of the neighbors and have a total amount, we need to average that to get the actual pixel value. We can do this by dividing the total by the amount of neighboring pixels we've looked at! Remember that our code almost draws a square around the current pixel we're at: we look at kernel_size in the up/down direction AND in the left/right direction. Below is an example if we had a kernel_size of 3 and were looking at pixel (1,1) (where x represents any pixel we would add to the total)

x	x	x	
x	Pixel	x	
x	x	x	

Now it's your turn:
Calculate the average_pixel value and store it into the new Image's current pixel. Outside of all of the loops, return the new Image!

Congrats! You have successfully blurred an image! If you want to test your image run the Image test case file and make sure that `high_blur`, `low_blur`, and `average_pixel` pass.

The `apply_kernel` Function

The last thing that we will be working on today is the `apply_kernel` function! We've actually technically already done this in our blur function: all applying a kernel does is that it multiplies each neighboring pixel we're using by a matrix, and then sums up all of those values to get the new Image's current pixel. Below is a picture made using a kernel that helps detect edges!

Abbildung 9: Original Image

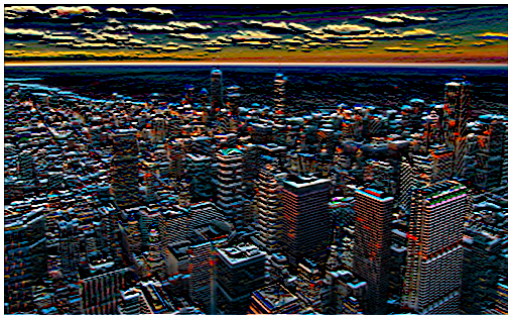


Abbildung 10: Edge lines are more vibrant!

> *Stop and think*: What type of matrix do you think we used in the blur function? Why?

Because we already implemented a kernel earlier in `blur`, we can copy all of the code that we did! The only thing that we're going to change is a few variables and the values that we will assign to the pixels.

> Now it's your turn:
> Paste your code from `blur` into `apply_kernel`. Delete the line that assigns the new Image pixel value to total though, as that is going to slightly change based on the matrix we pass in.

The `blur` function makes use of `kernel_size`, but you might have noticed that this is not a parameter for `apply_kernel`. We can fix this though by taking the shape of the kernel at any index!

> *Stop and think*: Why is kernel_size equal to the shape of the kernel at any index? Remember that a kernel is a matrix, and each row has the amount of operations you're supposed to do. It might help if you look at the example matrix from the blur function!

> Now it's your turn:
> At the very beginning of the code, extract the kernel's shape (this can be done like how you do it for an array!).

Great! We now have all of the starting variables we need. Inside all of our for loops, we now need to go through each neighbor pixel and apply every value of the matrix (remember: we have a total of 5 loops: the first three get us a specific pixel and the last two get us all of the neighbor pixels in the neighbor range).

This might seem tricky, but it's actually pretty straightforward! What we need to do is first make a variable for x and y that is centered around 0. What this means is that we need two variables that are going to be able to iterate through the kernel/matrix we are given, and extract every value out of them (ex: at [0,0] we have the value 1/3, at [1,0] we want to extract the value 2, and so forth!).

> *Stop and think*: Before we give one possible solution, how do you think this can be done? Remember that neighbor_range will be equal to the size of the matrix! Maybe you could do something with that value to find out how far you've gone from the current pixel location?

This can be done by declaring x_kernel = neighbor_pixel_x_location+ neighbor_range-x_value_of_actual_pixel.

If this doesn't make sense, all we're doing is seeing how far the current pixel is from one of its neighbors, and then adding the neighbor boundary to that! If this was the furthest neighbor pixel to the left, then (neighbor_pixel+neighbor_range)-x_value_of_actual_pixel will get you 0. This means that we are able to get the x_zero index of the matrix! If you want to get the first one, you would have to go to to the next neighbor pixel. Since this is closer to the last pixel, when we add neighbor range we're going to get an extra 1 (and therefore, the x_one index of the matrix)!

Stop and think: Why are we only saying the x_zero index of the matrix instead of just the zero index of it? (Hint: A matrix is a 2D array! What does that mean?)

Now it's your turn:
Create the x_kernel variable and the y_kernel variable. Note that the y_kernel variable code is almost exactly the same as the x_kernel variable (just some minor changes)!

We're almost done! The second to last thing we want to do is store our actual matrix value (using the indexes we got), and then multiply that by the current neighbor pixel we're looking at (remember that we're iterating through the neighbors of the pixel within the neighbor range currently)! What this is doing is directly applying whatever was in the matrix to the current neighbor (so if we had 1.2 in [0,0] of the matrix we would be multiply the neighbor by this amount). We then want to add this value to the total.

Now it's your turn:
Create a variable called kernel_value, which is set to equal whatever is in the kernel at the x and y indices you just got. Then, on the next line, add to total whatever is at the old Image's pixel multiplied by the kernel_value.

The very last thing we want to do is now update the new Image's pixel value! At the current pixel's location, set it equal to the total.

Now it's your turn:
Finally, set the current_pixel of the new Image equal to the total amount. This should be done outside of the neighbor pixels loop but inside the num_channel loop. Outside of all the loops, make sure to return the new Image!

If you want to try and see how this filter works, feel free to pass in a numpy array for the kernel parameter! Some examples to try are np.array([[1,2,1],[0,0,0],[-1,-2,-1]]) (a horizontal edge detector) and np.array([[1,0,-1],[2,0,-2],[1,0,-1]]) (a vertical edge detector). Feel free to look up more possible kernels you can use and try them out!

Challenge: Motion Blur

Sometimes when objects zoom by they seem to 'blur' from how fast they're traveling. Today we will be replicating this effect, using a stationary object as our subject! We can use a slightly modified `blur` function to accomplish this.

Stop and think: Examine the image above. Our earlier `blur` function completely blurs the image, which is not what we want in this case. We want our object to seem like it's moving in a singular direction, which means it has to be blurring that way. How can that be accomplished?

Abbildung 11: Original Image

Abbildung 12: Kiki speeding on her broom!!

Now it's your turn:
Create the `motion_blur` function! This should be extremely similar to the `blur` function, aside from how many (and what type) of pixels are blended together. The Stop and Think should help you determine what you need to change!

Checkoff - End of Part 2:
By the end of this part you should have

- Coded the `blur` function
- Coded the `apply_kernel` function
- Coded the Motion Blur Effect

Show a course staff member your working code to get the checkoff.

In the next part we will:

- Create the `combine_image` function
- Code the `invert_image` function
- Add a border to an image

In Part 3 we will:

- Create our `combine_images` and `inverted_images` function
- Give an Image a border!

Let's get to work!

The `combine_images` function

What does combining an image mean? One possibility is taking multiple objects from each image and then placing them both in a new Image (this is something you would typically see in photoshop!). Another alternative (and the one that we'll be implementing today) is overlapping two images at every single point so that we have an accurate representation of both throughout the image.

Stop and think: How do you think we could implement the first combining method suggested? What are some functions/objects that we would need to have access to before we could do this?

Let's start by creating an Image object! As always, we will be assigning the x,y, and num-channels values. We will also iterate through each one of them to get to every single pixel! Note that whatever Image you use to create the new Image object will not matter for this function, as we are assuming the two pictures are the exact same size.

Now it's your turn:
Create a new Image Object and store its x-values, y-values, and number of channels in their respective variables. Then, iterate through the x-pixels, y-pixels, and num-channels.

Once we've gotten to a specific pixel, we need to think about how we want to update it. We want to represent both values, which means we need to take equal weights from both of them. To accomplish this task, we should take a value that accounts for the 'weight' of each pixel, and is able to sum them together. As a result, we'll be using the Pythagorean Theorem to accomplish this!

Now it's your turn: Use the Pythagorean Theorem on Image1.shape.array[x][y][num-channel] and Image2.shape.array[x][y][num-channel] to get the value of the new Image! Then, return the Image!

Stop and think: How could we implement this if we wanted to combine images that were different sizes?

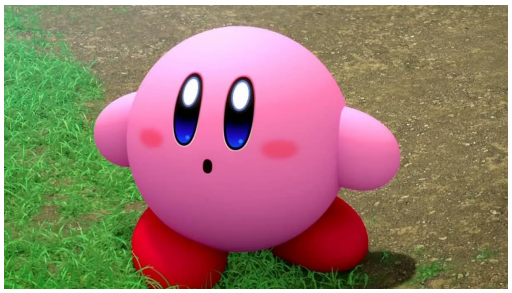


Abbildung 13: Original Image

Abbildung 14: Kirby in a field of flowers!!

Great job! You should be able to now produce an image that has two images overlaying each other. If you want to test your image run the Image test case file and make sure that combine_images pass.

The `invert_image` Function

The last official function that we will be making is the `invert_image` one!

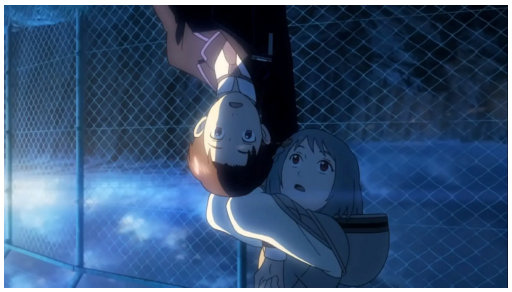


Abbildung 15: Who's upside down?

Stop and think: Before you move on, try to answer this question! To invert something is to make an object upside down. How do you think we can invert an image? (Hint: maybe it has something to do with the order we read the pixels!)

Prepare yourself: we're going to be reading an image backwards for this implementation! Like the *create_a_border* function we'll be making an additional count for the x and y values. The code is extremely similar to *create_a_border* so in fact, you can actually just copy the function's code up until the if statements that are comparing the dimensions! This code isn't perfect though: there are some things that we have to update to read the image backwards.

Now it's your turn: Copy the code from textitcreate_a_border and paste it in your `invert_image` function.

Stop and think: What are some things that you think are wrong with the current code? Remember that we are reading the image starting at the very end as opposed to the beginning like usual.

Note that *create_a_border* has the starting x and y value and start pixel. We don't want that though: In this case we want to start our counting at the very bottom left (since we read left to right). Think about what indexes those are, and update the values appropriately!

Now it's your turn: Change your starting values for the x and y counters (outside of the loop). Remember that we want to start at the bottom left corner. What values of x and y would that equal to?

Now that that's fixed, we also have to change how we're updating our variables! One of them is currently being updated wrong. Which one do you think it is?

Stop and think: In our original *create_a_border* function we were reading from left to right, top to bottom. Now in `invert_image` we're still reading left to right, but are now going from bottom to top. Which variable do you think we're going to be changing how we update? (Remember that up to down is rows and left to right represents columns).

Now it's your turn: Correct either x or y counter updater. One of them is currently increasing when it should be decreasing.

The last thing we're going to do is update the New Image's current pixel value. Remember, we are writing the new Image upside down: whatever is at the current pixel value (that comes from the x,y,num_channels for loop) should be equal to the new Image at the x_counter,y_counter, num_channel value.

Now it's your turn: Update the new Image's pixels appropriately!
Return the new Image!

Stop and think: For this function we made it so that the image was read upside down, but what if we wanted to flip the sides that the image was on instead? Note: This should almost be an opposite implementation to what we did today!

And with this, we're done! :D

Challenge: Creating a Border

Today our challenge is to make a border! Some of the code will be similar to the `combine_images` function, but there are definitely a few differences! In our implementation, the user will pass in two pictures, image1 and image2. Image 1 will be the picture that we want to add the border to, and image2 will be the picture that will function as the border. `create_a_border` also accepts a starting pixel, which tells us what part of the picture we want to take as the border from (this means that image2 can be a full picture, not just an empty frame!). We also have a parameter called 'dimensions', which accepts an array that has the how wide the border is (the top and bottom is from index 0 and the sides are from index 1). The default is that the width and length of the border will be whatever picture 1 has them as.

Stop and think: What do you think the dimensions of the border has to be, and what happens to the interior of it?

Abbildung 16: Original Image

Abbildung 17: Kiki in the clouds

For this challenge we will need to have two counters: one will be our regular for loop that will tell us where we are in the new image, and the other will be the one that starts at the starting pixel-1 we are given. To do this, we're going to initialize our x and y counter outside of the loop, so that we know where we want to start looking!

Now it's your turn: Initialize your x and y counter variables to the starting pixel, subtracting 1 to each of them. Then, copy all of the code in `combine_image`, excluding the line that uses the Pythagorean theorem.

Stop and think: Why are we starting at the starting pixel-1 instead of the exact amount? Isn't starting pixel a valid index?

Now we need to make sure that we're updating the counters that we initialized outside of the loop. Remember that these should be increasing whenever the main loops pixel values are (ex: if the main x loop just went up then the x counter will go up as well). We also need to check after we update the y counter and see if we have

checked all of the y pixels (in other words, if we have checked all of the columns for a specific row). Since this is a manual counter (with us updating it with +=1 instead of a for loop) we have to reset it.

Now it's your turn:
Increase the x and y counter by one wherever the x and y in the for loop are increasing. Also, as soon as you update the y counter make sure to check if it needs to be reset to 0. If it does, change it to 0!

Stop and think: What happens if we don't reset the y counter we created? Also, why is it OK if the border picture is larger than the actual picture we're using?

Finally, we actually want to paste our "border"(image2) into our new image! Here's how this is going to work:

- If we are within the top/bottom few rows the new Image will automatically use the borders pixels
- If we are within the most outer columns (on either side) we will use the borders pixels.
- Otherwise, we will use Image1's pixels to make the New Image

Abbildung 18: Start-pixel of [10,5] and dimensions of [30,50]

Here's an example to make this easier: If I passed in dimensions=[30,50], this would mean that I want the border to stretch from row 0, to row 30. I also want it to stretch from the very bottom to the bottom-30 (so that the bottom entire rows are covered). I've also passed in 50 in the column area, which means on the sides I want column 0 to 50 taken up by Image2/the border, in addition to the very right side-50. If I'm not at these values, then I should fill New Image with Image1's pixels! Look at the image above if you need help (for this Image I passed in two images, a start-pixel of [10,5], and dimensions of [30,50])!

Now it's your turn:
Implement the if statements and their consequences!

Outside of the loop, return the new Image!

You should now have a working `create_a_border` code! See if you pass `create_a_border` in the test case file!

This was the last official challenge we had so amazing job! You now know how to edit an Image in code (who needs PhotoShop anyway?)! Tomorrow we will make one last Image just to try out some of our functions!

Checkoff - End of Part 3:
By the end of Part 3 you should have:

- Coded the `combine_images` function
- Coded the `inverted_images` function
- Added a border to a picture using the `create_a_border` function

Show a course staff member your working code to get the checkoff.

The last part is completely optional, but there are a few fun challenges you can do!!

- Optional: Finish up the project by making one last final image! :)

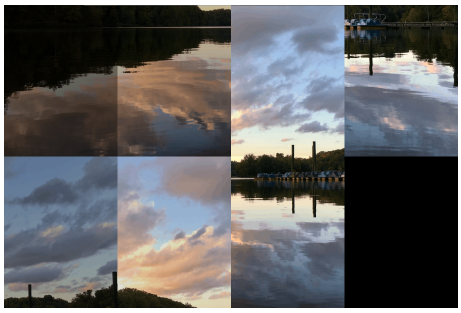
This last part is optional :) There are two things available here:

- The first is that you can try to code the 'Sliding Puzzle', which scrambles up pictures.
- The second is that you can make your own functions that manipulate photos! (I have some suggestions below if you want to try to implement them)

Let's get right into it!

Option 1: Challenge: Sliding Puzzle

Have you ever played a sliding puzzle? These are games where you're often given an image that has been rearranged, and the goal is to fix it by gradually moving around different parts of it. We will be trying to replicate this, scrambling up a few parts of our picture and then brightening them based on their original location. By the end of the challenge you should have a very interesting (and distinctive!) piece of art. Note that this is the hardest challenge, and the other ones are a lot easier so don't worry if you struggle a bit with this one!



The `scramble_image` Function

Before we begin, let's quickly go over how this function works. `scramble_image` should break an Image object into 8 quadrants, which are determined by the length and width of the png. Once those quadrants are determined, you then want to randomly select these quadrants (one by one) and put them in the new Image. We will be putting these pixels in from left to right, one quadrant at a time. Below is the order for how we will be filling in the new Image.

1	3	5	7
2	4	6	empty

Now that we've done an overview, let's do this one step at a time!

Initializing Variables

The first thing we want to do is initialize some variables! Like before, we are going to want to make a new Image object and then store those properties so we can access them. We're going to need a couple of additional variables though, as our approach for this project is going to be a lot different compared to before!

Stop and think: What are some variables/things that we're going to have to keep track of as we fill in the new image?

The first thing that we are going to need is to know how 'much' a section is. If we want to divide our picture up, we're going to need to know how long and wide each part of it is going to be. In the table above, our picture is divided so that our sliding puzzle is two sections tall and four sections wide. What would the length and height of each of our sections be?

> Now it's your turn: Create a new Image object. After creating it, make two variables, called length and height. These should represent the length and height of one section. How can you calculate this? (Look at the table if you get stuck!).You might also need to typecast this value :)

Great! Now that's done, we also have to consider how we want to place our sections in the new Image. Since we want to randomly select the order, we should probably use the random module! If you look at the top you can see that it has already been imported, which means you just have to use it. //
To do this, I recommend that you create a list that contains the starting pixel of each section (so a list that contains a list of pixel values)! The height and length value created earlier should help a lot here.

> *Stop and think*: Why do you think a list is practical/useful for this? Think about some of the properties/abilities of lists!

> Now it's your turn: Create a list of pixel values (there should be one for each section, so 8 in total).

There's only three more variables we need! Because we're reading an image and storing it into another randomly, we are going to need two additional counters. These counters will tell us where in the new Image we will be placing the section (while the for loop will be extracting the pixels from random sections). Our counter will allow us to start at [0,0], and then gradually add pixels from top to bottom. Since we're starting at [0,0], we should initialize both the x and y counters to -1 (this is because of how we update them later!). We're also going to need a variable that lets us know when we need to reset the y value, and only updates when we have finished reading all of the rows of two sections that are vertically aligned (in the table, for example, when 1 and 2 have been added to the new Image). Right now we can call this how_many_column_sections, and initialize this to 0.

> Now it's your turn: Create the x-counter, y-counter, and how_many_column_sections variable. Make sure to initialize them to their correct amounts!

We can now start to actually fill in the new Image!

Filling in the New Image

The first thing we need to do is create a loop that keeps putting in random images until 7 of them have been added (the last one will be an empty space). Next, we need to select a random section that we want to use to fill in the New Image. We also want to make sure that the section can't be used again.

> Now it's your turn: Make a loop that runs for 7 iterations. In that loop we should select a random sector (Hint: you can select from the list we made earlier) and then remove it so it can not be chosen again.

The rest of the code will also all be in the for loop we just made (excluding our return statement)! // Now, we need to extract the x and y coordinates of the sector we just chose (remember, these are currently stored in a list that stores list values). This is so that we can easily access the pixels later!

> Now it's your turn: Make a variable called x_start and y_start that let us know the x and y coordinate of the start of the sector we're using.

Now, like always, we're going to have to make three nestled loops to go through each pixel! This time though, we are going to have a slightly different range for our x and y loop. The x loop should only go from the start row of the sector, to the last row of the sector, while the y loop should go from the start column to the last column of that section. In each one of these loops you should also make sure to update the proper counter (in the x_loop you should update x_counter and in the y you should update y_counter). The reason why we have

this is because the main loop will help us go through the original Image (and select the random sections within range) and the counter will help us fill in the new sections (as the values start at 0 and are only increasing by 1! In other words, the counters allow us to fill in the new Image from left to right, and then top to bottom).

Now it's your turn: Create the 3 for loops and adjust the ranges as specified above. Also make sure to add the x_counter and y_counter in their proper place! They should increment by one every single time their loop starts over :)

We're making good progress! So far we have the accessing pixel code set up: now we need to change each of the pixels to their proper random area! Before we do that though, we have to think about some things. The way that we're used to reading the image is top to bottom, where we start at row 0 and read all of it before moving on to row 2. In this function though, we only want to read one section at a time, which means we only want to read a specific height and length of the original Image (not the entire thing!). To do that we're going to have to look at a few things:

- Have we checked too many columns of the original Image for a section?
- Have we read the last row in the original Image?

And with that... I leave it to you! This is a bit of a fun thought exercises and is a bit tricky, so you're welcome to come to office hours if you need help! There's also a hint in the Stop and Think below that will give you some variables to mess around with.

Stop and think: How can we check each of these conditions? (Remember that we have a counter for both x and y which keeps track of where we are in the new Image. We only want to keep placing pixels (for a specific random section) as long as we're in the correct sector! We also want to make sure that we're always in a valid place in the new Image. We don't want to go out of bounds!) Also, what do we do if these conditions are exceeded/met? (Another Hint: For example, if we've checked all of the columns we're supposed to in the first row, that probably means we need to reset the y_counter back to 0 so that we can go to row 2 and do the rest of the rows in the section. What about our x_counter though? When would we want to reset that back to 0?)

Now it's your turn: Check both conditions and update the x_counter and y_counter appropriately. Once that's done, set your new_Image[x_counter][y_counter][z] = original_image[x][y][z] (where z is the current channel you're on). Once you're done with that, outside of the loop return the new Image!

Great job on all your hard work! You should now have a scrambled image which will reset and change as many times as you call the function.

Stop and think: This current version removes one of the sections to reduce the amount of work we have to do. This can be a problem though if the last image is one of the middle pieces. If we had more time, how could we fix the code so that only end/side pieces would be the missing piece? What about if we wanted to code it so that all of the pieces fit (plus the empty space so the puzzle still worked)?

Option 2: Creating your own functions!

Here are some ideas:

- Make a grey-scale function
- Make a function that pixelates images
- Make a filter/effect that applies multiple filters we use! (Ex: Lomography uses contrast, blur, brightness (vignettes))
- Double/mirror an item in an image!

These are just some starting points though, feel free to create whatever you want! Great job on all your hard work! I hope you had fun learning how to manipulate Images in Python!