**Nicholas Bowman, Sonja Johnson-Yu, Kylie Jue** **Section #3**
**CS106AP** **July 9, 2019**

# Section Handout #3: Images, Lists, and Files

## 1. Image Puzzles

*This is a lab-style problem. You may want to work on this in groups on your laptop and test your programs as you write them.*

Photo Editor programs are often used for color-correcting digital images, that is, modifying the color components in images to alter the relative "redness," "greenness," or "blueness" of the image. In this problem, we'll give you three exaggerated examples of images that need color correction, but you'll need to figure out exactly is wrong with each of them. Once you've done this, you'll write code to alter each pixel in order to correct the image. For these problems, you should be able to use the plain for-each loop to solve each of them without needing to work with `x` and `y` coordinates.

Specifically, implement the following functions, each of which takes in an image filename and returns a SimpleImage. You might find the SimpleImage documentation useful to consult.

For the benefit of students who are color-blind or otherwise unable to identify issues with the colors of these images, we've listed the exact issues with each of the images in a table at the end of this problem. If you find it necessary to complete the functions, please feel free to refer to that table directly rather than experimenting with the different color components.

- `fix1(filename)`: This function takes in the `filename` of an image where all of the RGB values of each pixel have been darkened. For each of the red/green/blue components, one component has been divided by 2, one by 4, and one by 6, and it's your job to figure out which components have been divided by which numbers. The same channel will always be divided by the same factor. Write code to fix the image and then return it.
  *Note that the divide/multiply operations may introduce some glitches into the image due to rounding errors, but these can be ignored.*

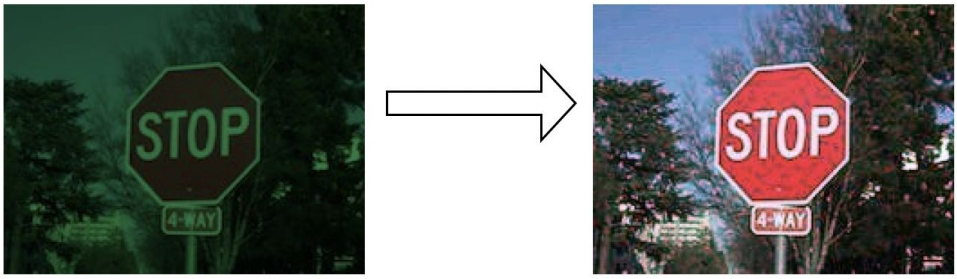


**Figure 1**: The input and output images for `fix1()`.

- `fix2(filename)`: This function takes in the `filename` of an image where two of the RGB color components have been swapped consistently across all of its pixels. Figure out which two colors have been switched and write code to fix it. This function does not suffer from divide/multiply glitches. Swapping is perfectly reversible!

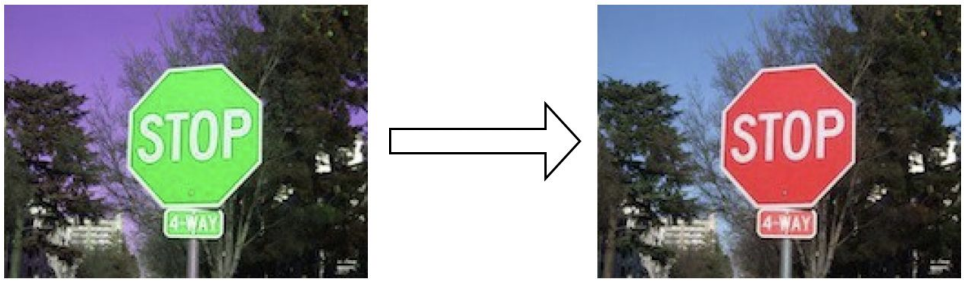


**Figure 2**: The input and output images for **`fix2()`**.

- **`fix3(filename)`**: This function takes in the **`filename`** of an image where two of the RGB color components have been swapped consistently across all of its pixels. In addition, the third color has been divided by 4. Write code to fix all of these modifications.

**Figure 3**: The input and output images for **`fix3()`**.

| Function | Issue with Image |
|---|---|
| **`fix_1()`** | In each of the original image's pixels, the red component has been divided by 6, the green component by 2, and the blue component by 4. |
| **`fix_2()`** | In each of the original image's pixels, the red component has been swapped with the green component. |
| **`fix_3()`** | In each of the original image's pixels, the red component has been swapped with the blue component, and the green component has been divided by 4. |

## 2. Image Range Loop Problems

You might find nested range loops helpful in solving these. Problems like this are well-suited to making a drawing first to think through the algorithm before coding and especially to get the arithmetic for each coordinate exactly right. These algorithms are a magnet for off-by-one errors.

A good first step is to sketch out the pixel grid **`x,y`** coordinates as in lecture (see Figure 4) to plan how your algorithm will access the pixels.

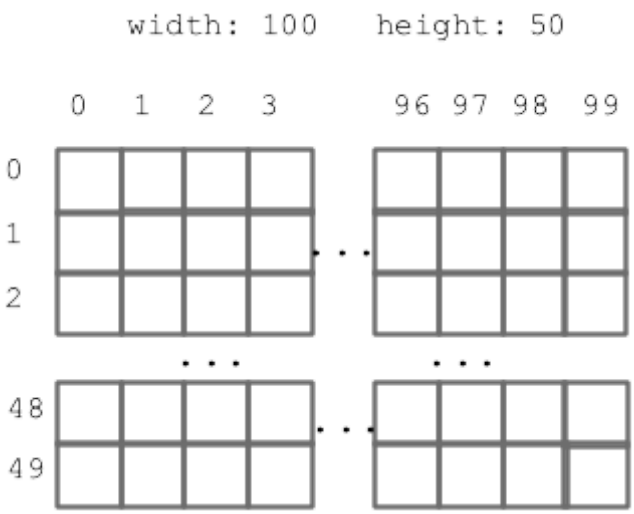


**Figure 4**: The x,y coordinate system for pixels in an image of width 100 pixels and height 50 pixels.

Implement the following functions, each of which takes an image's filename and returns a SimpleImage. You might find the SimpleImage documentation useful to consult.

- **`double_left(filename)`**: Takes the left half of the image, and copies it on top of the right half.

**Figure 5**: The expected output for **`double_left()`**.

- **`double_left_up(filename)`**: Takes the left half of the image, and copies it on the right half as before, except it is flipped upside-down as it is copied.

**Figure 6**: The expected output for **`double_left_up()`**.

- **`copies_2(filename)`:** Creates a new image twice as wide as the original, and places two copies of the original image in it, side-by-side.

**Figure 7**: The expected output for **`copies2()`**.

- **`squeeze_width(filename, n)`:** A funhouse mirror effect governed by the integer parameter **`n`**. Create a new image with the same height as the original, but with a width that is **`n`** times smaller than the original's. Copy the original image such that it is squeezed horizontally.

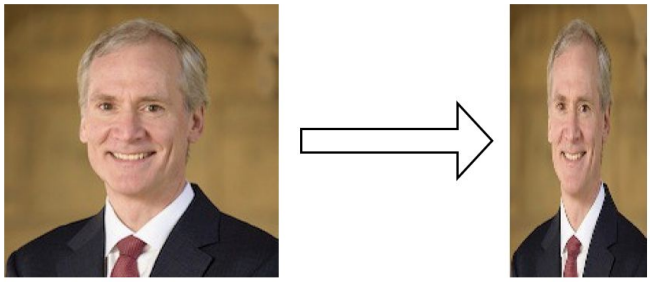

**Figure 8**: The expected output for **`squeeze_width()`**.

For example, if **`n = 4`**, the pixels in the output image with an **`x`** coordinate of **`0`** would be copies of the input pixels with an **`x`** coordinate of **`0 * 4 = 0`**, while pixels in the output image with an **`x`** coordinate of **`1`** would be copies of the input pixels with an **`x`** coordinate of **`1 * 4 = 4`**, and so on.

## 3. Photoshop Image Algorithms

- **Rotate left**
  Write the following function:

  **`def rotate_image_left(image)`**

  that takes a **`SimpleImage`** parameter called **`image`** and returns a version of that image, rotated 90 degrees to the left.

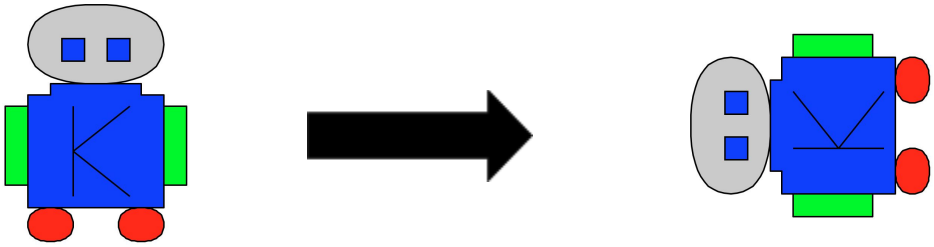

**Figure 9**: The expected output for **`rotate_image_left()`**.

Note that the source image's width may be different from its height and that the dimensions of the result image reflect the rotation you have performed. The resulting image is as wide as the source was tall, and as tall as the source was wide.

- **Flip vertical**
  Write the following function:

  `def flip_vertical(image)`

  that flips a `SimpleImage` parameter called `image` in the vertical direction.

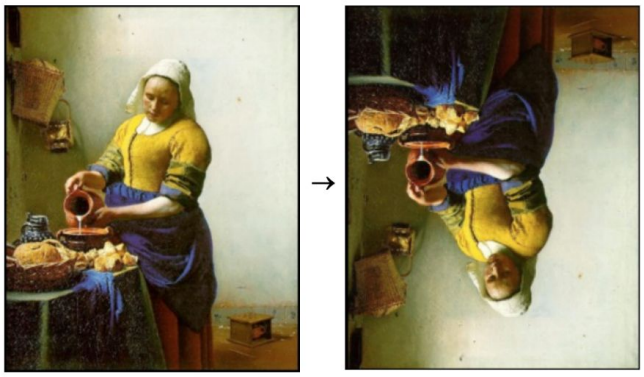

**Figure 10**: The expected output for `flip_vertical()`.

**Challenge**: Try to solve this problem without creating a new image. That is, do the image manipulation in place on the original image.

### 4. Range and List Problems

Here, we'll go through a series of problems intended to increase your familiarity with lists and with the additional parameters of the `range()` function. Implement the following functions:

1. `negative(n)`: Given a non-negative integer `n`, return a list of all the ints from `n` down to `-n`, i.e. `[n, n-1, ..., 1, 0, -1, -2, ..., -n]`.

   *Note: consider the case where* `n == 0`. *Does your function need to do any special work to account for this situation?*

2. `mirror(lst)`: Given a list `lst`, return a new list that is composed of all the elements of `lst` in original order, followed by the elements of `lst` in reverse order. That is, `mirror(['how', 'are', 'you'])` should return `['how', 'are', 'you', 'you', 'are', 'how'].`

3. `unique_numbers(nums)`: Given a list `nums`, return the number of unique numbers contained in `nums`. The values in the list are guaranteed to be in sorted order, which means that duplicates will be grouped together. However, there may or may not be duplicates in the list. If passed an empty list, the function should return 0. For example, `unique_numbers([5, 7, 7, 7, 22, 22, 23, 35, 35, 40, 40, 40])` should

return 6 since there are 6 unique numbers (5, 7, 22, 23, 35, and 40).

### 5. Calculator

Your job is to write a program that emulates the three calculator functions shown in Figure 11:

```
$ python3 calculator.py -square 42
1764 # prints the square of the number passed in

$ python3 calculator.py -exp 2 10
1024 # prints the first number raised to the power of the second number

$ python3 calculator.py -add 1 2 3 4 5
15   # prints the sum of all the numbers typed in
```

**Figure 11:** A sample output of running the three separate functions in your calculator program

You may assume that you are provided with a **`main()`** function that takes as a parameter the list of arguments typed in the console, as in Figure 12:

```
import sys

def main(args):
    # your code here
    pass

if __name__ == "__main__":
    main(sys.argv)
```

**Figure 12:** The general structure for your program, including a **`main()`** function with an args list as its parameter

Thus, your job is to decompose and implement the **`main()`** function so that your program produces the sample output above.

**Notes:**

- You may assume that the correct number of parameters are passed in to the program: specifically, **`-square`** will be followed by two numbers, **`-exp`** will be followed by two numbers, and **`-add`** will be followed by at least one number.
- You will need to deal with the fact that the command line parameters are passed to your program as strings, and you will need to convert them to ints.
- There are no run configurations included in the starter code for this file! **You are expected to run this program entirely from the PyCharm terminal.**

## 6. Word Count

Write a program WordCount that reads a file and reports how many lines, words, and characters appear in it. Suppose, for example, that the file **`lear.txt`** contains the following passage from Shakespeare's King Lear:

```
Poor naked wretches, wheresoe'er you are,
That bide the pelting of this pitiless storm,
How shall your houseless heads and unfed sides,
Your loop'd and window'd raggedness, defend you
From seasons such as these?  O, I have ta'en
Too little care of this!
```

Given this file, your program should be able to generate the sample run shown in Figure 13.

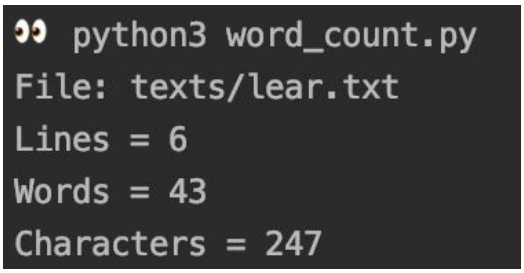


**Figure 13**: A sample run of **`word_count.py`** on **`lear.txt`**.

For the purposes of this program, a word consists of a consecutive sequence of characters delimited by spaces. You can assume the user will enter a valid filename.

**Note**: There are no run configurations included in the starter code for this file! You are expected to run this program entirely from the PyCharm terminal.