

Project 5: Pixel Magic

Purpose

To implement a program that accepts command-line arguments as well as tie together core concepts learned throughout the quarter.

You may not use any image utility modules available for Python.

Description

For this project, you will write functions that transform an image in the following ways:

1. Adjusting colors to reveal a hidden picture.
2. Darkening colors to produce a fading effect.
3. Apply a filter to an image to remove salt-and-pepper noises.

Download the image files from Canvas.

Image File Format

The P3 (.ppm) format is a text-based format (meaning that it is readable text) that defines an image as a sequence of pixels beginning with the top-left pixel and stored in row order (i.e. every pixel in a row is stored in left-to-right order and before any pixel in the next row).

A file conforming to the P3 format begins with header information. The header consists of the characters `P3`, the integer width of the image (in pixels), the integer height of the image, and the maximum value for a color component (we will use `255` for this value). Immediately following the header is the color information for each pixel. A pixel's color is represented by three integers denoting the red, green, and blue components (in that order). The following example shows how a file would look if it contained an image initially with one blue pixel, one red pixel, and one green pixel.

```
P3
3 1
255
0 0 255
255 0 0
0 255 0
...
```

The top-left pixel is said to be at location `<0, 0>` and the bottom-right at location at `<width - 1, height - 1>`.

Testing

You are required to write at least 3 tests using the unittest module for each helper function that takes an argument / arguments and returns a value (i.e. is not an I/O function). Since we are emphasizing test-driven development, you should write tests for each function first. In doing so, you will have a better understanding as to what the functions take as input and produce as output, which makes writing the function definitions easier. Create **pixelmagic_tests.py** and write your tests in the file.

Implementation

Your program must support the following three modes: 'decode', 'fade', and 'denoise'. In this project, you can decide how you want to decompose your program into functions. However, it is recommended that you create at least three following functions. In the functions below, pixels refers to a list of lists of integers, in which each inner list (representing a single pixel) contains 3 integers, one for each color value. Create **pixelmagic.py** and write your program in the file.

```
main()
```

```
find_image(pixels)
```

Returns decoded pixels. Corresponds to the filter mode 'decode'.

```
fade_image(pixels, width, row, col, radius)
```

Returns faded pixels. Corresponds to the filter mode 'fade'.

```
denoise_image(pixels, width, height, reach, beta)
```

Returns denoised pixels. Corresponds to the filter mode 'denoise'.

Error and Usage Messages

Regardless of the filter mode, print one of the following error messages when its corresponding condition is true.

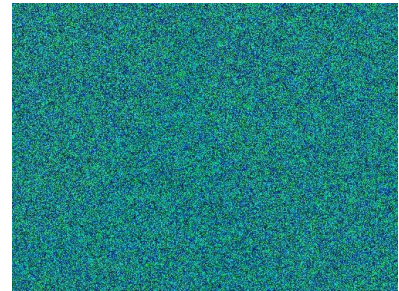
- If either the filter mode or image path is not provided, your program must print:
Usage: python pixelmagic.py <mode> <image>
- If the image argument is provided but cannot be opened (e.g. does not exist), your program must print (replacing the blank with the name of the file given):
Error: Unable to Open _____
- If the mode argument is not decode, fade, or denoise, your program must print:
Error: Invalid Mode

For all other messages to display, see the corresponding sections below.

1. Hidden Image: 'decode' mode (30 points. No partial credit.)

This program will get you started with the core functionality of processing a P3 image file. Your program must take a single command-line argument that specifies the name of the input image file. It will output the decoded image to a file named `decoded.ppm`. The output must be a valid P3 image file; do not forget to write the required header information to it.

The puzzle image hides a real image behind a mess of random pixels. In reality, the image is hidden in the red components of the pixels. Decode the image by increasing the value of the red component by multiplying it by 10 without allowing the resulting value to pass the maximum value of 255. In addition, set the green and blue components equal to the new red value. Shown below is the hidden image; it will be obvious once you have properly decoded the puzzle image.



2. Fade: 'fade' mode (30 points. No partial credit.)

This program is a relatively minor modification of the previous program. In addition to the filter mode and file name, it must take three additional command-line integer arguments (in the following order):

1. The row (y-coordinate) position of the fade center
2. The column (x-coordinate) position of the fade center
3. The fade radius

These three arguments may be assumed to be integers. If any of these arguments are not provided, an appropriate usage message must be printed to the terminal and the program should terminate. This message is as follows:

```
Usage: python pixelmagic.py fade <image> <row> <col> <radius>
```





Your program will output the faded image to a file named `faded.ppm`, which must be a valid P3 image file; do not forget to write the required header information to it.

This program will transform pixel values based on their distance from a specified point (this point may fall outside of the image). The row and column coordinates specified on the command-line give the point and the radius is used to control the fading.

For each pixel, compute the distance from the pixel location to the specified point. Scale (multiply) the color components of the pixel by:

$$(\text{radius} - \text{distance}) / \text{radius}$$

Do not use a scale value below 0.2, which prevents very dark borders around the image).

	
original image	row = 300, col = 450, radius = 300
	
original image	row = 100, col = 150, radius = 100

3. Noise Reduction: 'denoise' mode (up to 40 points. A partial credit possible.)

In 'denoise' mode, your program is required to apply the median filter to remove salt-and-pepper noise from a black-and-white image. The **median filter** is a non-linear digital filtering technique, often used to remove noise from an image or signal. The median filter removes noise by replacing a pixel with the median of its neighboring pixels within a window of a certain size.

In addition to the filter mode and file name, your program must take two additional command-line numerical arguments (in the following order):

1. An integer argument, "neighbor reach", to be used to determine the window of the filter (this argument is optional and will be used to determine which neighbor pixels to consider in the averaging calculation, as discussed below) Use a default value of 2 if the user does not specify it.

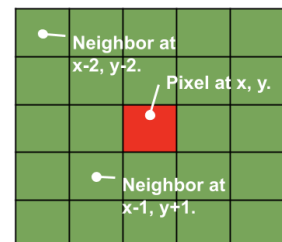
2. A floating-point number argument, "beta" value, to be used to determine if a pixel must be replaced or not. Use a default value of 0.2 if the user does not specify it.

Usage: `python pixelmagic.py denoise <image> <reach> <beta>`

Your program must output the filtered image to a file named `denoised.ppm`. You need to take a deep copy of the original pixels, then modify the copy. But you must use the original pixels to compute the median. You may use `copy.deepcopy()`: e.g. `new_pixels = copy.deepcopy(pixels)`. You need to import the `copy` module. The output must be a valid P3 ppm image file; do not forget to write the required header information to it.

Your program will filter the image by computing, for each pixel, the median of nearby pixels (more precisely, the median over one color component of the nearby pixels. It does not matter which color channel value you use because you are only required to filter a black and white image, but in the following explanation, we assume that you use the red channel. In a black and white image, all three channels have the same value.). A pixel's replacement color will be determined by the colors of the pixels within a specified "neighbor reach". The default "neighbor reach" is 2 (but this can be modified by a command-line argument as discussed previously). Thus, a pixel's replacement color will be determined by the colors of the pixels within two pixels to the left or right and within two pixels above or below (this will form a square window of size 5). The following diagram is meant to help illustrate.

This diagram shows how to compute the color for the pixel in the center (the red element). Its neighbors (within a reach of 2) are all of the green elements. The pixels outside of this 5x5 square are not considered in the replacement color calculation for this pixel. To compute the color for the center pixel, sort the red values of pixels in the window in ascending order. Some pixels will not have the full complement of neighbors (such as those on or near the edge of the image). In these cases, just find the median of the existing neighbors (i.e. those within the bounds of the image).



Suppose that the pixel values of neighbors are put in a list called `neighbors`:

```
neighbors = [123,225,231,...,45]
```

Then the list must be sorted first. **You may not use Python's builtin `list.sort()` method nor the `sorted()` function.** Implement either the selection sort or the insertion sort. You may copy your sorting function from Project 4 and paste it into `pixelmagic.py` file.

After sorting, the list will look as follows:

```
neighbors = [45,...,123,225,231]
```

Then you can pick the median pixel value with the following code:

```
med = neighbors[len(neighbors)//2]
```

The pixel's new RGB values shall be all set to the same value: `med`.

To avoid replacing a good pixel unnecessarily, you will use the beta value to determine if the pixel needs to be replaced by using the following formula:

if $\text{abs}(\text{original} - \text{median}) / (\text{original} + 0.1) > \text{beta}$, replace the original pixel with the median value. This is to ensure that only outlier pixels will be replaced. As you can see, a higher beta value leaves some noise but the image remains sharp.

		
original image with noise	reach = 1, beta = 0.15	reach = 2, beta = 0.1
		
reach = 2, beta = 0.15	reach = 2, beta = 0.2	

As you will notice, this process takes a very long time (more than 50 seconds). It is because the sorting algorithms we have learned in this course, selection sort and insertion sort, are not very efficient. If you have extra time to work on more challenging problems, you might want to implement a very efficient sorting algorithm called merge sort, <https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheMergeSort.html>.

Submission

Upload `pixelmagic.py`, `pixelmagic_tests.py`, and at least three generated images (.ppm files. One for each mode.) to Canvas.

The style of your code will be checked with `pylint` and up to 10 points might be docked from your grade.

You may not resubmit your work after being graded. So, make sure you run your program on your computer and test if it really works.