!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

CSE 160 Homework 3

Homework 3: Image Blurring

Due: at 11:59pm on Friday, October 30, 2020.
Submit your `blur_image.py` file via Gradescope. (REQUIRED survey)

Learning Objectives:

- gain experience writing functions and using lists in Python
- practice using loops and conditionals (if statements) in Python
- become familiar with reading and writing files in Python
- write Python code to blur an image

Advice from previous students about this assignment: 14wi 15sp

Background

You will use, modify, and extend a program to blur a black and white image using a simple 3 x 3 matrix.

What is Image Blurring?

One type of data that may not immediately seem amenable to manipulation via a Python program is image data. Yet that is exactly what you will do in this assignment! Large quantities of image data are collected every day: telescopes gather images of distant galaxies, satellites take photos of the earth's surface, your car license plate is photographed as you cross the 520-bridge, and video cameras collect footage as you enter a bank.

In this assignment you will write Python code to blur a black and white image. Blurring can be used to reduce the level of noise in an image and prepare it for further processing such as identifying features. Gaussian smoothing is one example of a blurring effect. We will do a very simple blur.

Blurring an image is a specific example of a applying a filter to an image. Many such filters can be applied by setting each pixel to a weighted combination of the neighboring pixels, called a convolution.

Contents:

The matrix of weights used is called the kernel of the transformation. If you are curious, Wikipedia has more information about image processing with convolution. That reading is optional and is not required to complete this assignment.

Your program will read in black and white images, and output a blurred version of that image. The grayscale images we will use can be thought of as a rectangular grid of pixels where each pixel is represented by an integer value from 0 to 255 (0=black, 255=white). Each location in the grid represents one pixel in the image. (Note that color images are represented differently - each pixel has 3 values, one each for redM`, green, and blue. The program you will write will not process color images, but we are giving you a Python program you can use to convert color images into black and white images so you can run the program on your own images if so desired.) The code that reads in an image and that writes out the correct image format has already been written for you.

The Algorithm

The simple blurring algorithm we will use takes each pixel value in the original grid and converts it into a new value to be placed in the same location in another grid (**we will not be modifying the original grid**). That new value will be calculated as the average of its original value and the original value of the 8 pixels that surround it. For example, if we are examining the pixel containing the value 3 in the image below, its average would be computed by adding all 8 surrounding values to 3 and then dividing the sum by 9. (We will do this using **truncating** integer division using the // operator in Python 3 - each pixel must be an integer between 0 and 255.) So the value stored in the new blurred grid would be (1 + 5 + 61 + 4 + 3 + 2 + 10 + 11 + 100)/9 or 197/9 or 21.

```
1    5   61
4    3    2
10   11  100
```

Your code will need to examine each location in the grid for the image you are given, and calculate the value for that same location in the new grid using the averaging approach described above. You might be wondering how you should compute this 9-value average for locations in the original grid that appear on the edge (or corner). For those locations, you should substitute the value 0 for any of the nine locations that "fall off the grid". So in the example above, if that 3 x 3 grid actually represented our entire image, we would calculate a new value for the upper right (northeast) corner (the value 61) as (0 + 0 + 0 + 5 + 61 + 0 + 3 + 2 + 0)/9 or 71/9 or 7. The result of blurring each image represented by the grid shown below on the left is shown on the right. Remember that we are using truncating integer division.

```
Original Image:          Blurred Image:
 1    2    3              1    2    1
```

```
 4   5   6              3   5   3
 7   8   9              2   4   3


Original Image:         Blurred Image:
 1  1  1  1  1  1       0  0  0  0  0  0
 1  1  1  1  1  1       0  1  1  1  1  0
 1  1  1  1  1  1       0  1  1  1  1  0
 1  1  1  1  1  1       0  0  0  0  0  0


Original Image:         Blurred Image:
 10  10  10  10         4   6   6   4
 10  10  10  10         6  10  10   6
 10  10  10  10         6  10  10   6
 10  10  10  10         6  10  10   6
 10  10  10  10         4   6   6   4
```

Contents:

Problem 1: Obtain the files, add your name

Obtain the files you need by downloading the homework3.zip file.

Unzip the `homework3.zip` file to create a `homework3` directory/folder. You will do your work here. Open the folder in VS Code. The `homework3` directory/folder contains:

- `blur_image.py`, a partial Python program that you will complete
- `images`, a directory which contains some sample black and white images for you to process.
- `test_grids`, a directory which contains some very small sample black and white images stored as text files in CSV (comma-separated values) format.
- `color_to_gray.py`, a Python program that will convert color images into black and white images.(This program is optional for you to use in converting your own images into black and white images and is described further in Problem 7.)

You will do your work by modifying `blur_image.py` and then submitting the modified version. **Add your name to the top of this file. Otherwise, there is no code to write for this problem.**

Problem 2: Run the program

Similar to homework2, for this assignment you will run your program by opening a **terminal window** in VS Code (if it's not already shown at the bottom of the screen, hit Ctrl+Shift+` while having a Python file open in an editor window). To run the program, type the following into the terminal window:

On Mac/Linux:

```
python blur_image.py images/Husky.png
```

On Windows:

```
python blur_image.py images\Husky.png
```

Note: If you get a "can't open file 'Husky.png'" error or a "No such file or directory" error, then perhaps you did not open up the `homework3` folder, or you mistyped the file name.

If you get a "ImportError: No module name Image" error, then you may have the wrong version of Python installed.

For now we are just checking that you have the right version of Python installed. **Once you have completed the assignment** (so NOT yet!), running the program as you just did should create two new files in your current directory:

- `Husky_blur.png`, a blurred version of the original Husky.png
- `Husky_blur_grid.txt`, a comma-separated values (CSV) file containing the integer contents of the blurred grid. This file will be useful for debugging your program. We have intentionally given it the file extension .txt (instead of the more common .csv) so that you can open it easily in a text editor .

The .png file can be opened in any image browser and should appear slightly blurred. (You can feed files to your program multiple times to blur them more.) The .txt file can be examined to determine if you are performing blurring correctly.

That sounds great! Read on to learn how you can make all of this happen...**There is no code to write for this problem.**

Problem 3: The Big Picture

Problem 3 asks you to take a look at several aspects of the program as a whole *before* you start writing any code. **There is no code to write for this problem.** You will probably want to read through this problem multiple times before proceeding to the next problem, since some of its descriptions rely on terms defined towards the end of the problem.

Take a look at `blur_image.py` focusing on the functions in the file (these all start with `def function_name`) and on the main program instructions listed at the end of the file. You might print this file out so you can make notes on it and so you can easily see all parts of the program at once. You will notice that this file is about 200 lines long. Don't worry - you do not need to understand all of the code in this file! We will walk you through the modifications you need to make in Problems 4-7 below. In total you will implement four short function

bodies (6-10 lines each) and add in some calls to those functions. For now let's just take a look at what we have given you.

The Main Program

Let's first take a look at the main program (look for the comment: "The main program begins here"). Read the comments in the code that describe what each piece of the given code does. The steps in the main program (see the comments: "Step A", "Step B", etc.) are roughly:

A. Process command line arguments - tell the user if they have not given the correct number of arguments to the program, otherwise get the name of the input_file from the user arguments.

B. Determine the type of the input_file based on its file extension (.txt, .png), and read the file contents into a grid of pixels stored as a list of lists of integers.

C. Generate the output file names - based on the name of the input_file, create the strings that will be used as the names for the output files. The program will write the blurred image to a file in two formats: as a .png image you can open in any image viewer and as a text file containing a comma separated list of the integer values in the grid, one row of the grid per line of the file. This text file will be useful for debugging your program.

Later, you will need to add code for the remaining two steps in the main program:

D. Apply the blur algorithm to the grid of pixels you read into the variable `input_grid` in step B.

E. Write the blurred grid of pixels to the two output file names you created in step C.

In order to complete steps D and E above you will need to call some of the functions in this file. Below is a brief guide to the functions in the file.

The Functions

Some of these functions are given to you and you do not need to modify them at all, some you need to implement.

Functions you are given to read and write files:

You do **NOT** need to modify these three functions. You do **NOT** need to know "How" these functions work, **you just need to understand "What" they do.**

- `read_image(file_path)` - Reads the image file at `file_path` into a rectangular grid of pixels, represented as a list of lists of integers. Each element of the outer list is one row of pixels, where each pixel is an integer x such that 0 <= x < 256. Returns the grid of pixels.
- `write_image(file_name, pixel_grid)` - Given `pixel_grid` as an image in a list of lists of integers format, write it to the

Contents:

filename `file_name` as an image.

- `write_grid(file_name, pixel_grid)` - Given `pixel_grid` as an image in a list of lists of integers format, write it to the filename `file_name` in CSV format.

Functions that you need to implement:

- `get_pixel_at(pixel_grid, i, j)` - Returns the pixel in `pixel_grid` at row `i` and column `j` (zero-indexed). Returns 0 if there is no row `i` or no column `j`, or if `i` or `j` are negative. Note: We are NOT allowing negative indexing into our `pixel_grids`.
- `average_of_surrounding(pixel_grid, i, j)` - Returns the (unweighted) average of the values of the pixel at row `i` and column `j` and the eight pixels surrounding it in the given `pixel_grid`.
- `blur(pixel_grid)` - Given `pixel_grid` (a rectangular grid of pixels), returns a **new** grid of pixels of the same size and shape, that is the result of blurring `pixel_grid`. In the output grid, each pixel is the (unweighted) average of that pixel and its eight neighbors in the input grid. Note: This function should create a new grid and return it. It should not modify the original grid.
- `read_grid(file_path)` - Reads the CSV file at `file_path` into a rectangular grid of pixels, represented as a list of lists of integers. This method should read any file written by the `write_grid` function. Returns the grid of pixels.

Other functions:

You will also notice a few more functions in the file. We will describe these in more detail later in Problems 4-7. You do **NOT** need to modify these three functions. You do **NOT** need to know "How" these functions work, **you just need to understand "What" they do.**

- `csv_line_to_list(line)` - Given a CSV-formatted row of `integers`, returns the integers as a `list` of `integers`. You will need to call this function in your implementation of `read_grid`.
- `test_get_pixel_at()` - Basic, brief sanity checks for `get_pixel_at`.
- `test_average_of_surrounding()` - Basic, brief sanity checks for `average_of_surrounding`.

The Data Structure

Note that several places above it refers to a "grid of pixels". This is the data structure we will use inside of our program to represent the image. As described in the Background section, the black and white images we are using can be thought of as a rectangular grid of pixels, where each pixel is represented by an integer between 0 and 255. Inside of our program we will represent this grid as a `list` of `lists` of `integers`. The first (outer) `list` will be a *list of rows* of the grid. The length of this list is the "height" of the grid. Each row of the grid will be represented as a

`list` of `integers`. Each row will be the same length. So the length of any one of these rows is the "width" of the grid. For example, the first image described in this writeup would be represented by the following list:

```
[ [1, 5, 61], [4, 3, 2], [10, 11, 100] ]
```

Here is another example pixel grid and the `list` that would represent it:

```
 1   2   3   4   5
 6   7   8   9  10
11  12  13  14  15

[ [ 1, 2, 3, 4, 5 ], [ 6, 7, 8, 9, 10], [ 11, 12,
13, 14, 15] ]
```

Assuming this list was assigned to the variable `grid`, to access the pixel in the top left corner (containing the value 1), you would say: `grid[0][0]`. To access the value in the bottom right corner (containing the value 15) you would say: `grid[2][4]`. (To access the 15 you could also say: `grid[len(grid)-1][len(grid[0])-1]`) Note: we will not allow accessing the grid using negative indexes (something that Python's `list` allows).

The File Formats

The program you are given will read in black and white image files in .png format. It will apply the blur algorithm to the given image and write the blurred image to two output files. One output file will be in .png format that can be opened in any image viewer. The other output file will be a text file (with file extension `.txt`) in CSV (Comma-Separated value) format. After you implement the `read_grid` function, your program will also be able to read input files in the same CSV format that the program produces. We have provided a few small sample files in this format in the `test_grids` folder. The purpose of supporting the reading and writing of files in this CSV format is to facilitate testing your program. While it might be hard to look at a blurred image in an image viewer and tell if you have done your calculations exactly right, this should be easy to do on small files in the CSV format. For example, the contents of the CSV file `small_grid.txt` and the CSV file containing the result of applying the blur algorithm to that grid are shown below. Note the commas between elements.

```
small_grid.txt:
  0,   0,   0
  0,   9,   0
  0,   0,   0

small_grid_blur_grid.txt:
1,   1,   1
1,   1,   1
1,   1,   1
```

Contents:

Feel free to create your own text files in this same CSV format for testing purposes. **There is no code to write for this problem.**

Problem 4: Implement the Blurring Algorithm

For this problem, you need to implement the three function bodies related to blurring an image: `get_pixel_at(pixel_grid, i, j)`, `average_of_surrounding(pixel_grid, i, j)`, and `blur(pixel_grid)`. We suggest you implement them in that order. You might go back to the description of the functions to refresh your memory. Below are some specifics about each of these functions and testing them. We are having you implement these particular functions because they demonstrate a nice decomposition of the overall problem of blurring an image. **If you find that you are not calling all of these functions somewhere in your code you should go back and look for opportunities to do so.** In particular, `blur` should call `average_of_surrounding`, and `average_of_surounding` should call `get_pixel_at`. Note that the smallest possible `pixel_grid` is one that contains a single pixel (e.g. `[ [1] ]`). You can assume that you will never be given a `pixel_grid` smaller than this.

1) **`get_pixel_at(pixel_grid, i, j)`** should return the pixel in `pixel_grid` at row `i` and column `j` (zero-indexed) or 0 if there is no row `i` or no column `j`. **We will not allow negative numbers to be valid indexes into our grid.** This function can be done in about 4-6 lines of code, although it is fine if yours is slightly longer.

The function `test_get_pixel_at()` can be used to do some basic sanity checks to test that you have implemented `get_pixel_at` correctly. `test_get_pixel_at()` makes use of the `assert` statement (described on slide 34 in the lecture slides about functions).

`test_get_pixel_at()` creates a test pixel grid, and then executes a sequence of assert statements. `assert` statements assert things that should be **true** at that point in the program, such as `get_pixel_at(test_grid, 0, 0) == 1`. If any of the assertions in `test_get_pixel_at()` fail then an error message will be printed. If you see such a message this means that you have not implemented `get_pixel_at` correctly. Look at the `test_grid` in `test_get_pixel_at` and the message printed and then see if you can figure out what is wrong with your implementation of `get_pixel_at`.

Notice that there is already a commented out call to `test_get_pixel_at()` immediately after its definition. **Un-comment the call to `test_get_pixel_at()` now.** The way our program is wrtten, as soon as `get_pixel_at` and `test_get_pixel_at` have been defined, the function `test_get_pixel_at()` will be called. Try running

Contents:

your program as described above and pay close attention to notice if any test cases fail.

2) **`average_of_surrounding(pixel_grid, i, j)`** should return the average of the values of: the pixel at row `i` and column `j` and the eight pixels surrounding it in the given `pixel_grid`. This is the algorithm described in the Background section. `average_of_surrounding(pixel_grid, i, j)` will calculate this average for a single pixel. Notice how in the code we have given you we expect you to sum the nine pixels into the variable `pixel_sum` and then divide that sum by 9 using truncating integer division with the Python 3 `//` operator. You should not change this part of the code; this is how we intend for the average to be calculated. You will probably need to add anywhere from 6-10 lines of code to this function. It can be done using loops by adding in about four lines of code but for this homework, using loops is not required. If you feel stuck, first try writing the code without any loops, then, see if you can modify your code to use loops instead. It will be fine if you solution does not use loops and adds closer to 10 lines of code.

The function `test_average_of_surrounding` is similar to `test_get_pixel_at()`. It can be used to do some basic checks to test that you have implemented `average_of_surrounding` correctly. **Un-comment the call to `test_average_of_surrounding()` now.** Try running your program as described above and pay close attention to notice if any test cases fail.

3) **`blur(pixel_grid)`** - Given `pixel_grid`, a rectangular grid of pixels, `blur` should return a **new** grid of pixels of the same size and shape, that is the result of blurring `pixel_grid`. You will read the origial `pixel_grid` and write your results into a **new** grid. For each location in the given `pixel_grid`, compute its average based on values in `pixel_grid` and then store that average in the same location in the new grid you are creating. When you are done with this you should return the new grid. You will probably need to add anywhere from 8-15 lines of code to this function. **We strongly recommend using the approach of starting with an empty list and appending things onto lists to create your new grid, as other approaches (e.g. copying grids) are likely to result in hard to track down bugs**.

****** See this Python Tutor example that shows why copying is NOT what we recommend!**.

Here is a sample `pixel_grid` and the new blurred grid that should be returned.

```
pixel_grid:
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

new blurred grid:
[[1, 2, 1], [3, 5, 3], [2, 4, 3]]
```

There is no test function provided for `blur`. Although you may write one if you wish, using the two test functions above as models. Writing a test function for blur is not required but we encourage you to get in the practice of writing test functions. You can find other examples of blurred grids elsewhere in this writeup. You can examine the output of your blurring algorithm visually after you have finished Problem 5 below. You won't know whether you have blurred the image exactly according to the algorithm until after you have finished Problem 6 and can compare the output of small blurred test_grids.

After you have implemented `blur` and are convinced that `get_pixel_at` and `average_of_surrounding` are working properly, move on to Problem 5.

Problem 5: Write the Main Program Code

For this problem you must replace the two comments in the main program with code that will implement step D (apply the blur algorithm) and step E (write the results to two output files). You might go back to the discussion of the main program to take a closer look at that we are asking. You will want to call several of the functions described above. This should consist of a small number of lines of code (3-5 lines total).

Nice work, you have finished the main program! You should be able to read in .png images, blur them, and write the result to a .png and a .txt file. Try it out!!!

Problem 6: Implement Reading CSV Grids

Finally, **for this problem you need to implement `read_grid(file_path)` to allow your program to read in files in our CSV format.** Note that you should NOT use any methods from Python's CSV module to implement this function: everything you need to know can be found in the lecture slides on File I/O.

Your code for `read_grid` should first open the file, and then read the file one line at a time. You should use the provided function `csv_line_to_list(line)` to convert each line of input into a `list` of `integers`. You will want to create a `list` of these lists that matches the "grid of pixels" format. Finally, you should close the file and return the grid of pixels.

If you go back and look at the main program you will notice that we are already calling `read_grid` where we need to. So you do not need to add in any calls to `read_grid`. Now you should be able to call

`blur_image.py` giving it *either* a `.png` image or a file with the `.txt` extension in our CSV format.

Problem 7: Checking your work

Before submitting, we recommend that you confirm that your code passes the tests in `test_get_pixel_at()` and `test_average_of_surrounding`. In addition, you should test that your CSV grid reading and blurring process works by reading in the files in the `test_grids` folder and comparing your output grids to the corresponding blurred grid in `test_grids`.

For example, if you run:

On Mac/Linux:

```
python blur_image.py test_grids/medium_grid.txt
```

On Windows:

```
python blur_image.py test_grids\medium_grid.txt
```

This should produce a file called `medium_grid_blur_grid.txt` in your `homework3` directory. You can use the Diff Checker tool from HW2 to compare YOUR file to the file called `medium_grid_blur_grid.txt` in the `test_grids` directory.

Note:

- The images represented by the files in the `test_grids` folder are very small! If you try to view the image they represent it will only show up as a tiny dot or a black screen depending on how you are viewing the image. Clearly you cannot tell if a tiny grid was blurred properly by looking at the image. You will want to compare the produced .txt file to see if it matches the one in `test_grids` as described above.
- If you open up a grid file in an editor be careful not to modify it. In particular, an empty line, even at the very end of the file will cause our program to raise an error.
- Creating other simple grids and calculating their results for comparison is also a good approach. You can use a spreadsheet program to calculate the contents of the blurred grid by dragging the formula across a grid.

You may also enjoy blurring some of your own images. Although since your program will only accept black and white images you may first need to convert your images to black and white. We have provided `color_to_gray.py`, a simple Python program that converts color images into black and white .png images. (Use of this program is optional.) To use the program, in either Mac/Linux or Windows type:

```
python color_to_gray.py MyImage.jpg
```

where `MyImage.jpg` is the name of your image (your image can be named anything but should have an appropriate file extension). This will create a new file in your current directory called `MyImage_BW.png` which can then be passed to `blur_image.py`. **There is no code to write for this problem, unless your testing uncovers a bug that you need to go back and fix :-).**

Submit your work

You are almost done!

At the bottom of your `blur_image.py` file, in the "Collaboration" part, state which students or other people (besides the course staff) helped you with the assignment, or that no one did. **This is required! Do not leave this blank!**

Submit the following files via Gradescope.

- `blur_image.py`

Answer a REQUIRED survey asking how much time you spent and other reflections on this assignment.

Now you are done!