Carleton University
SYSC 1005 – Introduction to Software Development – Fall 2015
Lab 11 - Image Filters that Use Lists, Tuples and Dictionaries

Objective

This lab reviews several topics that you need to know for the final exam; namely, image processing, lists, tuples and dictionaries.

Attendance/Demo

To receive credit for this lab, you must demonstrate the code you complete. **Also, you must submit your lab work to cuLearn by the end of the lab period**. (Instructions are provided in the *Wrap Up* section at the end of this handout.)

When you have finished all the exercises, call a TA, who will review the code you wrote. For those who don't finish early, a TA will ask you to demonstrate whatever code you've completed, starting about 30 minutes before the end of the lab period. **Any unfinished exercises should be treated as "homework"; complete these on your own time, before the final exam.**

Getting Started

1. Download Cimpl.py and the three image (JPEG) files that you used for Labs 4 through 7. Download solarize_lookup.py, convolution_filter.py, and nearest_color_filter.py. Download Lab_11_modified_images.zip. This is a compressed (zipped) folder that contains the images produced by my solutions to Exercises 2, 3 and 5. Use these images to help you verify that the images produced by your filters are correct.

2. Launch Wing IDE 101. Check the message Python displays in the shell window and verify that Wing is running Python version 3.4. If another version is running, ask a TA for help to reconfigure Wing to run Python 3.4.

Exercise 1

Open solarize_lookup.py in Wing IDE. That file contains the `solarize` and `build_solarize_lookup_table` functions that were presented in class. Read the code. Make sure you understand how the lookup tables are initialized, and how the solarizing filter uses the lookup table that is passed to it.

From the shell, load an image and call `solarize` three times. Observe how using a different lookup table each time changes the amount of solarization:

```
>>> img = load_image(choose_file())
>>> solarize(img, solarize_64_table)
>>> show(img)

>>> img = load_image(choose_file())
>>> solarize(img, solarize_128_table)
```

```
>>> show(img)

>>> img = load_image(choose_file())
>>> solarize(img, solarize_196_table)
>>> show(img)
```

Exercise 2

The `solarize` function uses a pixel's red, green and blue component values as indices when accessing the lookup table. The values obtained from the table are the integer components of the pixel's new (solarized) colour:

```
red = solarize_table[red]
green = solarize_table[green]
blue = solarize_table[blue]
```

We can also build lookup tables that store `Color` objects instead of integer component values. You'll use one such table in this exercise.

Hot metal is an image processing effect in which each pixel's red and green components are modified so that shades of red and yellow are dominant. (The RGB colour yellow is a mixture of red and green, and corresponds to the triplet (255, 255, 0)).

Here's a description of the effect. For each pixel in the image, we first calculate a *weighted brightness* for the pixel, using the formula:

weighted_brightness = 0.3 * *red* + 0.59 * *green* + 0.11 * *blue*

(This differs from the commonly used formula for calculating a pixel's brightness,

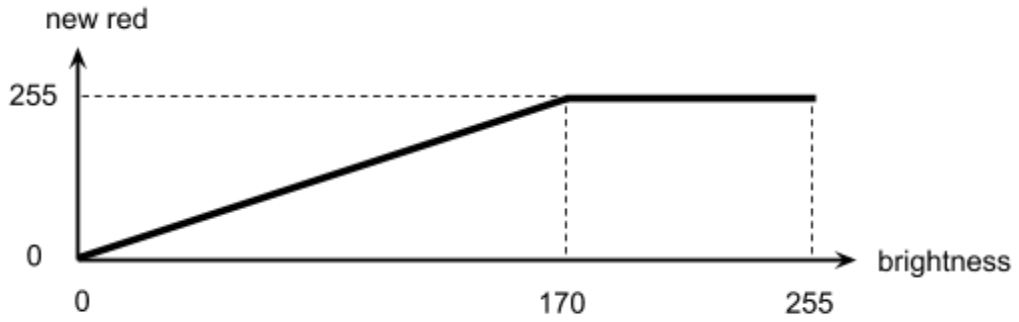
(*red* + *green* + *blue*) / 3

in that it emphasizes the green component and deemphasizes the blue component.)

We then convert this weighted brightness into an integer, and use that value as an index into a lookup table containing `Color` objects. The colour obtained from `table[weighted_brightness]` is the pixel's new colour.

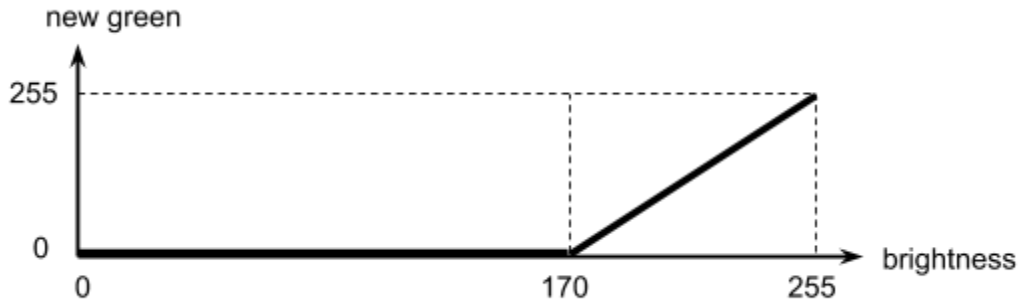
Building the hot metal lookup table is the interesting part of this exercise. This table contains `Color` objects created by `Cimpl`'s `create_color` function, and not integer component values.

The red component of each colour in the lookup table is defined by this graph:



If a pixel's brightness is between 170 and 255, the red component is of its new colour is 255. If a pixel's brightness is between 0 and 170, the red component of its new colour is the value that lies on the straight line between (0, 0) and (170, 255).

The green component of each colour in the lookup table is defined by this graph:



If a pixel's brightness is between 0 and 170, the green component of its new colour is 0. If a pixel's brightness is between 170 and 255, the green component of its new colour is the value that lies on the straight line between (170, 0) and (255, 255).

The blue component of the new colour is always 0.

Here's an example of how one `Color` in the table is created. Suppose a pixel's brightness is 170. From the graphs, we see that the corresponding red component is 255 and the corresponding green component is 0. So, we initialize location 170 in the lookup table with the colour (255, 0, 0). When the hot metal filter is executed, if a pixel's weighted brightness is 170, the replacement colour obtained from the table is (255, 0, 0).

Here's another example: suppose a pixel's brightness is 255. From the graphs, we see that the corresponding red component is 255 and the corresponding green component is 255. So, we initialize location 255 in the lookup table with the colour (255, 255, 0). When the hot metal filter is executed, if a pixel's weighted brightness is 255, the replacement colour obtained from the table is (255, 255, 0).

Step 1: Open a new file in Wing IDE and save it with the name hotmetal_lookup.py. Define a function named `build_hot_metal_lookup_table`. The function header is:

```
def build_hot_metal_lookup_table():
```

This function creates and returns a lookup table containing 256 `Color` objects, with the RGB values calculated using the approach described earlier. Hint: use a loop to "step through" all of the brightness values (0, 1, 2, ..., 255). For each brightness level, calculate the values of the new red and green components (you have to derive the formulas from the graphs), create the new colour, and put the colour to the table.

Use the shell to interactively call your function, and inspect the table it returns. Are the colours in the table correct?

Step 2: Add this statement to hotmetal_lookup.py, immediately after the definition of your `build_hot_metal_lookup_table` function:

```
hot_metal_table = build_hot_metal_lookup_table()
```

Be careful with your indentation! Make sure this statement isn't inside the function. As long as this statement is placed outside of the function, it will be executed every time your module is loaded into the Python interpreter and bind `hot_metal_table` to a newly created lookup table.

Step 3: In hotmetal_lookup.py, define a function named `hot_metal` that has two parameters: an `Image` object and the lookup table returned by your `build_hot_metal_lookup_table` function. The function header is:

```
def hot_metal(img, table):
```

For each pixel in the image, the function calculates the pixel's weighted brightness and replaces the pixel's colour with the colour obtained from the lookup table.

Interactively test your hot metal filter:

```
>>> img = load_image(choose_file())
>>> hot_metal(img, hot_metal_table)
>>> show(img)
```

Exercise 3

Convolution kernels and their application to image processing were presented in a recent lecture. A brief discussion of the technique can be found here:

http://en.wikipedia.org/wiki/Kernel_(image_processing)

Step 1: Open convolution_filter.py in WIng IDE 101. This file contains a filter named `convolution_filter`. There's also a test function that calls the filter, passing it a kernel for blurring images. Read the comments at the beginning of the file that explain the convolution algorithm, and read the code.

Different effects can be obtained by passing different kernels to `convolution_filter`. Here are the matrices for five different kernels:

blur

```
1  1  1
1  1  1
1  1  1
```

sharpen

```
 0   -3   0
-3   21  -3
 0   -3   0
```

emboss

```
-18  -9   0
-9    9   9
 0    9  18
```

edge detect 1

```
0    9   0
9  -36   9
0    9   0
```

edge detect 2

```
-9  -9  -9
-9  72  -9
-9  -9  -9
```

Step 2: In convolution_filter.py, define a function named `build_kernel_table` that returns a dictionary that has been initialized with the five 3-by-3 kernels shown earlier. The dictionary keys are strings containing the names of the filters; for example, `"blur"`, `"sharpen"`, etc. The value associated with each key is a kernel, which should be implemented

as a tuple containing three tuples (for an example, look at the initialization of `blur_kernel` in convolution_filter.py).

Use the shell to interactively verify the dictionary returned by `build_kernel_table`. For example, the following statements will retrieve and display the blur and sharpen kernels:

```
>>> kernels = build_kernel_table()
>>> kernel = kernels["blur"]
>>> kernel
((1, 1, 1), (1, 1, 1), (1, 1, 1))

>>> kernel = kernels["sharpen"]
>>> kernel
((0, -3, 0), (-3, 21, -3), (0, -3, 0))
```

Step 3: Modify function `convolution_filter` so that it is called with three arguments: an image, the table (dictionary) of kernels returned by `build_kernel_table`, and the name of a kernel. The revised function header is:

```
def convolution_filter(img, kernels, name):
    """ Return a new image created from the picture bound to img,
    using the specified 3-by-3 convolution kernel.
    Parameter kernels is a dictionary of convolution kernels.
    Parameter name is the name of the kernel (a string) to apply
    to the image.
    """
```

Modify the filter so that it retrieves the specified kernel from the dictionary, then applies it to the image. Interactively test your `convolution_filter` function. For example, to blur an image, we call `convolution_filter` this way:

```
>>> kernels = build_kernel_table()
        ...
>>> new_image = convolution_filter(img, kernels, "blur")
>>> show(new_image)
```

Verify that your modified `convolution_filter` works with all the kernels in the dictionary.

Wrap-up

1. Remember to have a TA review your solutions to the exercises, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the demo/sign-out sheet.

2. Before you leave the lab, log in to cuLearn and submit hotmetal_lookup.py and your modified version of convolution_filter.py. To do this:

    2.1. Click the Submit Lab 11 link. A page containing instructions and your submission status will be displayed. After you've read the instructions, click the

Add submission button. A page containing a File submissions box will appear. Drag hotmetal_lookup.py to the File submissions box. Repeat this for convolution_filter.py. **Do not submit files with a different name** (if one of your modules has a different name, rename it before submitting it). **Do not submit another type of file (e.g., a zip file, a RAR file, a .txt file, etc.)**

2.2. After the icons for both files appear in the box, click the Save changes button. At this point, the submission status is "Draft (not submitted)". If you're ready to finish submitting the files, jump to Step 2.4. If you aren't ready to do this; for example, you want to do some more work on the code and resubmit it later, you can leave the file with "draft" submission status. When you're ready to submit the final version, you can replace or delete your "draft" file submission, by following the instructions in Step 2.3, then do Step 2.4.

2.3. You can replace or delete a file by clicking the Edit my submission button. The page containing the File submissions box will appear.

2.3.1. To overwrite a file you previously submitted with a file having the same name, drag another copy of the file to the File submissions box, then click the Overwrite button when you are told the file exists ("There is already a file called..."). After the icon for the file reappears in the box, click the Save changes button.

2.3.2. To delete a file you previously submitted, click its icon. A dialogue box will appear. Click the Delete button., then click the OK button when you are asked, "Are you sure you want to delete this file?" After the icon for the file disappears, click the Save changes button.

2.4. Once you're sure that you don't want to make any changes, click the Submit assignment button. A Submit assignment page will be displayed containing the message, "Are you sure you want to submit your work for grading? You will not be able to make any more changes." Click the Continue button to confirm that you are ready to submit your lab work. This will change the submission status to "Submitted for grading".

Extra Practice Exercises

Exercise 4

For this exercise, go to the OPT Web site (http://pythontutor.com/visualize.html#mode=edit). Configure OPT to use Python 3.3.

The *distance* between two integers *a* and *b* is the absolute value of their difference; that is |*a* - *b*|.

Define a function named `find_nearest_integer` that has two parameters: a list of integers, `lst`, and an integer value, `val`. The function returns the integer in `lst` that is nearest to `val`; that is, it determines which integer in the list has the shortest distance to `val`. The function

header is:

```
def find_nearest_integer(lst, val):
```

Try these test cases (type these statements below your function definition, then click the Visualize Execution button and step through the execution of your function):

```
nearest = find_nearest_integer([5, 3, 9, 0, 6], 6)  # returns 6

nearest = find_nearest_integer([5, 3, 9, 0, 6], -3) # returns 0

nearest = find_nearest_integer([5, 3, 9, 0, 6], 8)  # returns 9

nearest = find_nearest_integer([5, 3, 9, 0, 6], 7)
# returns either 5 or 6
```

Don't delete your function. You may want to refer to it while you work on Exercise 5.

Exercise 5

Step 1: Open nearest_color_filter.py in Wing IDE.

This module contains statements that create a bunch of `Color` objects, followed statements that create six *colour palettes* (lists of colours). For example, `palette_1` is bound to a list containing `Color` objects for the RGB colours black (0, 0, 0), white (255, 255, 255) and gray (128, 128, 128). Because these statements are defined outside of any function, they are executed every time nearest_color_filter.py is loaded into the Python engine.

An RGB colour can be considered to be the Cartesian coordinates of a point in three-dimensional space; for example, gray can be thought of as the point (128, 128, 128). The distance between two colours can be calculated as the Euclidian distance between the two colour "points".

The `Cimpl` module defines a function that returns the Euclidian distance between two `Color` objects. Here it is:

```
def distance(color1, color2):

    r1, g1, b1 = color1
    r2, g2, b2 = color2

    return math.sqrt((r1 - r2) ** 2 + (g1 - g2) ** 2 +
                     (b1 - b2) ** 2)
```

In nearest_color_filter.py, define a function named `find_nearest_color` that has two parameters: a `Color` object and a palette. The function header is:

```
def find_nearest_color(color, palette):
```

The function determines which `Color` object from the list of colours in the palette is nearest to `color`, and returns that colour. Your function should call `Cimpl`'s `distance` function to determine how near two colours are to each other. Hint: the algorithm is similar to the one you wrote for Exercise 4.

Step 2: In nearest_color_filter.py, define a function named `nearest_color` that has two parameters: an `Image` object and a palette. The function header is:

```
def nearest_color(img, pallette):
```

The function modifies the image so that it contains only those colours in the specified palette. Each pixel's colour is changed to the colour from the palette that is nearest to the pixel's current colour.

Your filter must call your `find_nearest_color` function to determine the replacement colour for each pixel.

Use the shell to interactively test your function. For example, to modify an image so that the only colours are black, white, red, green, blue, cyan, magenta, and yellow, we call the function this way:

```
>>> img = load_image(choose_file())
>>> nearest_color(img, pallette_2)
>>> show(img)
```

Try all six palettes. (Remember to reload the original image before you call `nearest_color` with another palette; otherwise, the filter will modify an image that has already been modified.)