Carleton University
Department of Systems and Computer Engineering
SYSC 1005 - Introduction to Software Development - Fall 2018

Lab 5 - More Image Processing: Filters that Selectively Modify Pixels

Objectives

- Develop the first iteration of a module containing *image filtering* functions for a photo-editing application.
- Learn the syntax and semantics of Python's `if`, `if-else` and `if-elif-else` statements, by developing functions that selectively modify pixels in images.

Demo/Grading

After you finish all the exercises, a TA will review your solutions, ask you to demonstrate some of them, and assign a grade. For those who don't finish early, a TA will grade the work 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 your next lab.**

Getting Started

Step 1: Create a new folder named Lab 5. (If you're using a lab computer, you can create the folder on the desktop or in your account's Documents folder, or in the M: drive on the network server.)

Step 2: Download filters.py from the *Lab 5* folder on cuLearn. Move this file to your Lab 5 folder.

Step 3: Download Cimpl.py and the image (JPEG) files from the *Lab Materials* section of cuLearn. Move these files to your Lab 5 folder.

Step 4: Launch Wing IDE 101 and verify that it's running Python version 3.7.

Step 5: Open filters.py in a Wing IDE editor window.

General Requirements

All the functions that you develop must have a docstring containing:

- the function's type contract,
- a brief description of what the function does, and
- an example of how we can interactively test the function from the shell.

See the Python files containing the filters developed during recent lectures for examples of docstrings for image processing functions.

Exercise 1 - Improving the Grayscale Conversion Algorithm

This exercise introduces an important software engineering technique: after we design, code and test a function, we *refactor* it to improve it; for example, make it more understandable and maintainable, increase its efficiency, or improve the results it produces.

The `grayscale` filter in filters.py was developed in a recent lecture. It converts each pixel's colour to a shade of gray by using this formula to calculate the pixel's *brightness* (the average of its red, green and blue components):

```
brightness = (red + green + blue) // 3
```

and setting all three components to this average.

This calculation is (almost) equivalent to this formula:

```
brightness = red × 0.3333 + green × 0.3333 + blue × 0.3333
```

In other words, the red, green and blue components are weighted equally when calculating the average, with each component contributing 33-and-a-third percent.

This way of calculating grayscale does not take into account how the human eye perceives brightness, because we perceive blue to be darker than red. A better way of averaging the three components is to change the weight of the red component (to 29.9%), the green component (to 58.7%) and the blue component (to 11.4%):

```
brightness = red × 0.299 + green × 0.587 + blue × 0.114
```

In filters.py, make a copy of the `grayscale` function (copy/paste operations are provided in Wing's Edit menu.) Change the copy's name to `weighted_grayscale`. **Don't modify the original grayscale filter.**

Edit `weighted_grayscale` so that, for each pixel, it calculates the weighted average of the pixel's red, green and blue components and uses that value to create the shade of gray for the pixel.

Use the shell to test `weighted_grayscale`. Compare the images produced by this function with the images produced the original `grayscale` function. In your opinion, which grayscale filter produces better-looking images?

Exercise 2 - Extreme Contrast (Using `if-else` Statements)

A simple way to maximize the contrast between pixels is to change the red, green and blue components of each pixel to their minimum or maximum values. If a component's value is between 0 and 127, the component is changed to 0. If a component's value is between 128 and 255, the component is changed to 255. Use these values to create the pixel's new colour.

- How many different colours could there be in an image that has been modified by this filter? Be prepared to explain your answer to a TA.

In filters.py, define a function that returns a copy of an image in which the contrast between the pixels has been maximized. The function header is:

```
def extreme_contrast(image):
    """ (Cimpl.Image) -> Cimpl.Image

    Return a copy of image, maximizing the contrast between
    the light and dark pixels.

    >>> image = load_image(choose_file())
    >>> new_image = extreme_contrast(image)
    >>> show(new_image)
    """
```

When defining this function, use `if-else` statements. Do not use `if` statements or `if-elif-else` statements.

Use the shell to test `extreme_contrast`.

Exercise 3 - Sepia Tinting (Using `if-elif-else` Statements)

I'm sure you've seen old black-and-white (actually, grayscale) photos that, over time, have gained a yellowish tint. We can mimic this effect by creating sepia-toned images.

In filters.py, define a function that returns a copy of an image in which the colours have been sepia-tinted. The function header is:

```
def sepia_tint(image):
    """ (Cimpl.Image) -> Cimpl.Image

    Return a copy of image in which the colours have been
    converted to sepia tones.

    >>> image = load_image(choose_file())
    >>> new_image = sepia_tint(image)
    >>> show(new_image)
    """
```

There are several different ways to sepia-tint an image. In one of the simplest techniques, the image is converted to grayscale, then each pixel's red and blue components are adjusted, leaving the green component unchanged. Here's the algorithm:

- First, we convert the image to grayscale, because old photographic prints were grayscale. (To do this, `sepia_tint` should call your `weighted_grayscale` function. Don't replicate the grayscale algorithm in `sepia_tint`.) After this conversion, each pixel is a shade of gray, so its red, green and blue components are equal.

- Next, we tint each pixel so that it's a bit yellow. In the RGB system, yellow is a mixture of red and green. To make a pixel appear slightly more yellow, we can simply decrease its blue component by a small percentage. If we also increase the red component by the same percentage, the brightness of the tinted pixel is essentially unchanged. The amount by which we change a pixel's red and blue components depends on whether the pixel is a dark gray, a medium gray, or a light gray.

    - If the pixel's RGB components are less than 63, the pixel is in a shadowed area (it's a dark gray), so the blue component is decreased by multiplying it by 0.9 and the red component is increased by multiplying it by 1.1. (Hint: your function only has to compare one of the pixel's components to 63 - and not all three - because all three components in a shade of gray have the same value.)

    - If the pixel's RGB components are between 63 and 191 inclusive, the pixel is a medium gray. The blue component is decreased by multiplying it by 0.85 and the red component is increased by multiplying it by 1.15.

    - If pixel's RGB components are greater than 191, the pixel is in a highlighted area (it's a light gray), so the blue component is decreased by multiplying it by 0.93 and the red component is increased by multiplying it by 1.08.

When defining this function, use `if-elif-else` statements. Do not use `if` statements or `if-else` statements.

Use the shell to test `sepia_tint`.

Exercise 4 - Range Checking

RGB components have values between 0 and 255, inclusive. When we divide the range 0..255 into four equal-size quadrants, the ranges of the quadrants are 0..63, 64..127, 128..191, and 192..255. The process of determining the quadrant in which a component lies is known as *range checking*.

In filters.py, define a *helper function* named `_adjust_component` that is passed the value of a pixel's red, green or blue component. (Yes, the function's name begins with an underscore. A convention followed by Python programmers is to use a leading underscore to indicate that a function is intended for internal use only; in other words, that it should only be called by functions in the same module.) The function determines the quadrant in which a component lies, and returns the midpoint of that quadrant. Here is the function header:

```
def _adjust_component(amount):
    """ (int) -> int

    Divide the range 0..255 into 4 equal-size quadrants,
    and return the midpoint of the quadrant in which the
    specified amount lies.

    >>> _adjust_component(10)
    31
    >>> _adjust_component(85)
    95
    >>> _adjust_component(142)
    159
    >>> _adjust_component(230)
    223
    """
```

Note: the midpoints of the four quadrants are 31, 95, 159 and 223. These are the only values that this function returns.

Your function should assume that the integer bound to `amount` will always lie between 0 and 255, inclusive; in other words, it does not need to check if `amount` is negative or greater than 255.

Important:

- the argument passed to this function must be an `int`, not a `Color` object or an image;
- the value returned by this function must be an `int`, not a `float` or a `str`;
- this function does not print anything; i.e., it must not call Python's `print` function.

Interactively test your function, starting with the test cases shown in the function's docstring. These test cases use one value picked from each of the four quadrants, but four tests aren't sufficient. We should also have test cases that check values at the upper and lower limits of each quadrant. Interactively test your function to verify that:

- `_adjust_component(0)` and `_adjust_component(63)` return 31;
- `_adjust_component(64)` and `_adjust_component(127)` return 95;
- `_adjust_component(128)` and `_adjust_component(191)` return 159;
- `_adjust_component(192)` and `_adjust_component(255)` return 223.

Finish this exercise before your attempt Exercise 5.

Exercise 5 - Posterizing an Image

Posterizing is a process in which we change an image to have a smaller number of colours than the original. Here is one way to do this:

Recall that a pixel's red, green and blue components have values between 0 and 255, inclusive. We divide this range into four equal-size quadrants: 0 to 63, 64 to 127, 128 to 191, and 192 to 255. For each pixel, we change the red component to the midpoint of the quadrant in which it lies. We do the same thing for the green and blue components.

In filters.py, define a function that returns a posterized copy of an image. The function header is:

```
def posterize(img):
    """ (Cimpl.Image) -> Cimpl.Image

    Return a "posterized" copy of image.

    >>> image = load_image(choose_file())
    >>> new_image = posterize(image)
    >>> show(new_image)
    """
```

Your `posterize` function must call the `_adjust_component` function you wrote for Exercise 4 to determine the quadrant in which a component lies and return the value that is in the middle of the quadrant.

Use the shell to test `posterize`.

Wrap-up

When you've finished all the exercises, your `filters` module will contain 5 filters: `grayscale`, `weighted_grayscale`, `extreme_contrast`, `sepia_tint` and `posterize`.

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

2. Remember to backup your project folder before you leave the lab; for example, copy it to a flash drive and/or a cloud-based file storage service.

3. Any unfinished exercises should be treated as "homework"; complete these on your own time, before Lab 6.

4. You'll be adding functions to your `filters` module during Lab 6. Remember to bring a copy to your next lab session.