Carleton University
Department of Systems and Computer Engineering
SYSC 1005 - Introduction to Software Development - Fall 2017

Lab 5
Incremental Development of an Image Processing Module
Second Iteration: Filters that Selectively Modify Pixels

Objectives

- Develop the second 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

When you have finished all the exercises, call a TA, who 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.**

Prerequisite

Complete the Lab 5 prelab exercises before attempting this lab.

Getting Started

Step 1: Create a new folder named Lab 5.

Step 2: Download Cimpl.py and the image (JPEG) files from cuLearn to your Lab 5 folder. Copy the filters.py file that contains your solutions to the prelab exercises to your Lab 5 folder.

Step 3: Launch Wing IDE 101. Check the message Python displays in the shell window and verify that Wing is running Python version 3.6.

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.

Exercise 1

Step 1: Open filters.py in a Wing IDE editor window. This module should contain the `grayscale`, `solarize`, `black_and_white` and `black_and_white_and_gray` filters that were presented during recent lectures, plus your solutions to the prelab exercises (the `weighted_grayscale` and `negative` filters). Don't delete these functions!

Step 2: Click the Run button. This will load the module into the Python interpreter and check it for syntax errors.

Step 3: Read the definition of `solarize` (the header, the docstring and the function body). Using the approach you learned in Lab 4, call `Cimpl` functions from the Python shell to select an image file and load it into memory. Call `solarize` to modify this image, then display the modified image.

Repeat this step for the `black_and_white` and `black_and_white_and_gray` filters.

Exercise 2 - Modifying the Solarizing Filter (`if` Statements)

Currently, `solarize` compares each RGB component to 128. This number was chosen as the threshold because it's halfway between the largest and smallest component values (0 and 255).

Change this filter so that each component is changed only if its value is less than a specified integer threshold value. You will need to edit the function header, adding a parameter named `threshold` and changing the docstring as shown here:

```
def solarize(image, threshold):
    """ (Cimpl.Image, int) -> None

    Solarize image, modifying the RGB components that
    have intensities that are less than threshold.

    Parameter threshold is in the range 0 to 256, inclusive.

    >>> image = load_image(choose_file())
    >>> solarize(image, 128)
    >>> show(image)
    """
```

Use the shell to interactively test `solarize` with threshold values 64, 128 and 192. (Remember to reload the original image each time before you call the function; otherwise, you'll be modifying an image that has already been solarized.) What effect does increasing the threshold have?

Predict what the modified image will look like if 0 is passed as the threshold value. Call `solarize` with this argument, and see if your prediction was correct.

What threshold value will cause `solarize` to create the same effect as your `negative` filter?

Try it!

Exercise 3 - Extreme Contrast (Using `if-else` Statements)

Review the definition of the `black_and_white` filter. Make sure you understand how the `if-else` statement determines whether a pixel's colour will be changed to black or white.

In filters.py, define a function that is passed an image and maximizes the contrast between pixels. The function header is:

```
def extreme_contrast(image):
    """ (Cimpl.Image) -> None

    Modify image, maximizing the contrast between the light
    and dark pixels.

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

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 new component 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.

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 4 - Sepia Tinting (Using `if-elif-else` Statements)

Review the definition of the `black_and_white_and_gray` filter. Make sure you understand how the `if-elif-else` statement determines whether a pixel's colour will be changed to black or white or gray.

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 is passed an image and transforms it by sepia-tinting it.The function header is:

```
def sepia_tint(image):
    """ (Cimpl.Image) -> None

    Convert image to sepia tones.

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

There are several different ways to sepia-tint an image. One of the simplest approaches involves modifying each pixel's red and blue components, leaving the green component unchanged. The pixel's shade of gray determines the amount by which the red and blue components will be changed. Here's the algorithm:

- First, we convert the image to grayscale, because old photographic prints were grayscale. (To do this, `sepia_tint` must call your `grayscale` function. Don't replicate the grayscale algorithm in `sepia_tint`.) After this conversion, each pixel is a shade of gray; that is, 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. (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 5 - 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.) 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
    """
```

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 midpoints of these quadrants are 31, 95, 159 and 223, respectively. So, if `amount` is between 0 and 63, inclusive, the function returns 31.

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.

This exercise continues on the next page.

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, using 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 6.

Exercise 6 - 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. Suppose we divide this range into four equal-size quadrants: 0 to 63, 64 to 127, 128 to 191, and 192 to 255. We can use the _adjust_component function you wrote for Exercise 5 to determine the quadrant in which a component lies and return the value that is in the middle of the quadrant.

In filters.py, define a function that is passed an image and posterizes it. The function header is:

```
def posterize(img):
    """ (Cimpl.Image) -> None

    "Posterize" the specified image.

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

For each pixel, change the red component to the midpoint of the quadrant in which it lies. Do the same thing for the green and blue components..

Interactively test your posterizing function.

Wrap-up

When you've finished all the exercises, your `filters` module will contain 9 filters:, `grayscale`, `black_and_white`, `black_and_white_and_gray`, `weighted_grayscale`, `negative`, your modified `solarize` function, `extreme_contrast`, `sepia_tint` and `posterize`.

1. Remember to have a TA review your solutions to Exercises 2-6, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the attendance/grading 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.