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

Lab 6
Iterative, Incremental Development of an Image Processing Module
Third Iteration: Working with Pixels in a Range

Objectives

- Develop the third iteration of a module containing *image filtering* functions for a photo-editing application.
- Learn advanced techniques for manipulating digital images; e.g., using nested loops to generate pixel coordinates, and changing individual pixel colours based on the colours of neighbouring pixels.

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.**

Getting Started

Step 1: Create a new folder named Lab 6.

Step 2: Download blur_filter.py, Cimpl.py and the image (JPEG) files from cuLearn to your Lab 6 folder. Copy the filters.py file that contains your solutions to the Lab 5 Prelab and Lab 5 exercises, to your Lab 6 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 0 - Warm-up: Experiments with Nested Loops

Step 1: Open filters.py in a Wing IDE editor window. Don't delete the functions you developed

during Lab 5!

Step 2: Open blur_filter.py in a Wing IDE editor window. Copy/paste the definition of the `blur` function from blur_filter.py to filters.py. (You don't need to copy/paste the test functions.) Close blur_filter.py.

Step 3: Save your modified filters.py, then click the Run button to load the module into the Python interpreter. If necessary, correct any syntax errors.

In this lab, you'll be writing filters that use nested loops to generate the coordinates of individual pixels. A common error is generating coordinates that lie outside the borders of an image. When `get_color` is called with `x` and `y` arguments that represent invalid pixel coordinates, a run-time error will occur, and the filter will stop running.

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

Step 5: Edit `blur`, changing the `for` loops from:

```
    for y in range(1, get_height(source) - 1):
        for x in range(1, get_width(source) - 1):
```

to:

```
    for y in range(0, get_height(source) - 1):
        for x in range(1, get_width(source) - 1):
```

(The required changes are highlighted in **boldface**. You're changing the first argument in the call to `range` in the outer loop.)

Use the shell to test `blur`. What error message does Python display? What were the coordinates of the pixel that `get_color` was attempting to access when the error occurred?

Step 6: Edit `blur`, changing the `for` loops to:

```
    for y in range(1, get_height(source)):
        for x in range(1, get_width(source) - 1):
```

(You're changing both arguments in the call to `range` in the outer loop.)

Use the shell to test `blur`. What error message does Python display? What were the coordinates of the pixel that `get_color` was attempting to access when the error occurred?

Step 7: Edit `blur`, changing the `for` loops to:

```
    for y in range(1, get_height(source) - 1):
        for x in range(0, get_width(source) - 1):
```

(You're changing the second argument in the call to `range` in the outer loop, and the first argument in the call to `range` in the inner loop.)

Use the shell to test `blur`. What error message does Python display? What were the coordinates of the pixel that `get_color` was attempting to access when the error occurred?

Step 8: Edit `blur`, changing the `for` loops to:

```
    for y in range(1, get_height(source) - 1):
        for x in range(1, get_width(source)):
```

(You're changing both arguments in the call to `range` in the inner loop.)

Use the shell to test `blur`. What error message does Python display? What were the coordinates of the pixel that `get_color` was attempting to access when the error occurred?

Step 9: Edit `blur`, changing the `for` loops to:

```
    for y in range(1, get_height(source) - 1):
        for x in range(1, get_width(source) - 1):
```

(You're changing the second argument in the call to `range` in the inner loop.)

Use the shell to test `blur`. Verify that the function now runs without causing a run-time error.

Step 10: Read the code once more, and make sure you understand why changing the arguments passed to `range` caused invalid pixel coordinates to be generated.

Exercise 1 - Edge Detection

Edge detection is a technique that results in an image that looks like a pencil sketch, by changing the pixels' colours to black or white.

In filters.py, define a function that is passed an image and transforms it using edge detection. The function header and docstring are:

```
def detect_edges(image, threshold):
    """ (Cimpl.Image, float) -> None

    Modify image using edge detection.

    >>> image = load_image(choose_file())
    >>> detect_edges(image, 10.0)
    >>> show(image)
    """
```

Remember to include the entire docstring as part of the function definition.

A simple algorithm for performing edge detection is: for every pixel that has a pixel below it, check the *contrast* between the two pixels. If the contrast is high, change the top pixel's colour to black, and if the contrast is low, change the top pixel's colour to white.

One way to calculate the contrast between two colours is to calculate the brightness of the top pixel (the average of the red, green and blue components, with all three components weighted equally), and subtract the brightness of the pixel below it. We then calculate the absolute value of this difference. If this absolute value is greater than the filter's threshold parameter, the contrast between the two pixels is high, so we change the top pixel's colour to black; otherwise, the contrast between the two pixels is low, so we change the top pixel's colour to white.

Hint: your function needs to process one row of pixels at a time, and for each pixel, it will need to access the pixel below it. Don't use the

```
for x, y (r, g, b) in image:
```

pattern that you used in your other filters. Instead, use nested `for` loops to generate the (*x*, *y*) coordinates of pixels, and call `get_color` to get the `Color` object for each pixel. Remember what you learned in Exercise 0, and think carefully about the arguments that are passed to `range`.

Interactively test your edge detection filter. When calling the function, use 10.0 as the threshold. Repeat your tests with different thresholds. What range of threshold values result in images that look good to your eyes?

Exercise 2 - Improved Edge Detection

Make a copy of your `detect_edges` function, and rename this copy `detect_edges_better`. Don't delete or modify your original edge detection filter. Edit the docstring so that it looks like this:

```
def detect_edges_better(image, threshold):
    """ (Cimpl.Image, float) -> None

    Modify the image using edge detection.

    >>> image = load_image(choose_file())
    >>> detect_edges_better(image, 10.0)
    >>> show(image)
    """
```

Edit `detect_edges_better` so that it checks the contrast between each pixel and the pixel below it *as well as the pixel to the right of it*. As before, we calculate the contrast of two pixels by subtracting the brightness values of the pixels and calculating the absolute value of the difference. We change a pixel's colour to black only if the contrast between the pixel and the one below it is high (i.e., the absolute value of the difference exceeds the filter's threshold attribute)

or the contrast between the pixel and the one to the right of it is high. Otherwise, we change the pixel's colour to white.

Interactively test your function, using 10.0 as the threshold. In your opinion, does this filter do a better job of edge detection than the one you developed for Exercise 1?

Exercise 3 - Changing the Blurring Filter

Currently, the `blur` function calculates the component values for the new colour of each blurred pixel by averaging the components of the pixel, the pixel above it, the pixel below it, the pixel to the left of it, and the pixel to the right of it.

A better way to create the new colour for each blurred pixel is to average the pixel's red, green and blue components with the corresponding components of the 8 pixels that surround it; i.e., the new colour is based on the colours of 9 pixels.

Modify `blur` to do this. Think carefully about the arguments that should be passed to `get_color` to obtain the colours of each pixel and its eight neighbours.

Interactively test your blurring filter. In your opinion, does this filter do a better job of blurring than the one provided to you?

Wrap-up

When you've finished all the exercises, your `filters` module will contain 12 filters:, `grayscale`, `black_and_white`, `black_and_white_and_gray`, `weighted_grayscale`, `negative`, `solarize`, `extreme_contrast`, `sepia_tint` and `posterize` (from Lab 5), and `detect_edges`, `detect_edges_better`, and the modified `blur` function.

1. Remember to have a TA review your solutions to Exercises 1-3, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the 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 7.

4. You'll be using your `filters` module during Lab 7. Remember to bring a copy to your next lab session.

Challenge exercises start on the next page.

Challenge Exercise - Flipping an Image

In filters.py, define a function that is passed an image and transforms it by flipping it through an imaginary vertical line drawn through the center of the image. The function header and docstring are:

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

    Flip image around an imaginary vertical line
    drawn through its midpoint.

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

For example, if you had an image of a person looking to the right, the flipped image would show the person looking to the left.

You'll need to use nested loops.Within each row, the colour of a pixel located at some distance from the left side of the image must be swapped with the colour of the pixel located at the same distance from the right side of the image. This exercise is presented as a challenge, so I'm not going to provide the formulas that determine the (x, y) coordinates of the pairs of pixels that will have their colours swapped. That's something you have to figure out. Before writing any code, spend some time sketching diagrams and deriving the formulas. Hint: the body the inner loop is short (less than 10 lines of code).

Challenge Exercise - Another Flipping Filter

Define a function named `flip_horizontal` that is passed an image and transforms it by flipping it through an imaginary horizontal line drawn through the center of the image. You should be able to reuse much of the code from your `flip_vertical` filter.