Portfolio 2 Task: Image Watermarking (Medium difficulty)

Overview

This task involves developing two Python functions that manipulate image files by adding and revealing secret 'watermarks' using the Python Imaging Library. The watermarks are greyscale images which we embed invisibly inside larger full-colour images.

The goal of this task is to define two Python functions that operate on image files:

1. Function `add_watermark` takes a colour image and a greyscale watermark and hides the watermark in the image.
2. Function `reveal_watermark` takes a colour image with an embedded greyscale watermark and processes it so that the watermark is clearly visible.

The images and watermarks appear in files in Microsoft Bitmap format ('.bmp').

Digital image formats

Digital image files contain huge amounts of data, some of which can be changed or removed without making a significant difference to the overall picture. 'Lossy' image formats such as JPEG (Joint Photographic Experts Group) files exploit this property to compress images. (However, too much compression results in a loss of detail, visible as large uniformly-coloured squares.) 'Loss-free' image formats such as BMP (Microsoft Bitmap) files preserve the full detail of an image, but take up a lot of space. In this task we will use bitmapped images, because they have a comparatively simple format.

Digital images consist of a matrix of picture elements, called *pixels*, each forming a single dot. For instance, the following full-colour image file, `Flowers.bmp`, is supplied as part of this task.

If we greatly magnify this image, its top-left corner can be seen to be formed from individual pixels as shown below.

The colour of each pixel is defined by three 'channel' values, red, green and blue. Varying the brightness of each channel produces different colours. In the BMP image format the value of each channel is represented by a number in the range 0 to 255, inclusive, with larger numbers being brighter. For instance, the pixels in the enlarged image above are defined by the following red-green-blue tuples, starting from the top-left corner:

(159, 53, 0) (153, 43, 0) (159, 46, 0) (173, 76, 14) (188, 114, 33) (203, 142, 52)
(203, 139, 44) (178, 103, 18) (138, 55, 0) (124, 35, 0) (123, 35, 4) (152, 67, 31)
(195, 121, 68) (215, 149, 74) etc

Notice that as the colours get brighter, going from orange to yellow, the corresponding numbers get larger, going from around 160 to over 200 for the red channel. In this example the values for the blue channel are very low, often zero. At the extreme ends of the spectrum, tuple (0, 0, 0) is black and (255, 255, 255) is white.

Greyscale images are similar except that only one channel is needed, representing the brightness of the pixel. For example, the following greyscale image, `laboratory_watermark.bmp`, is supplied as part of this task.

If we greatly magnify this image, its top-left corner can be seen to be formed from individual pixels as shown below.

The brightness of each pixel is defined by a single value in the range 0 to 255, inclusive, with larger numbers being brighter. For instance, the pixels in the enlarged image above are defined by the following values, starting from the top-left corner:

113 113 112 110 109 108 107 107 98 106 107 97 93 97 100 98

88 82 69 61 etc

(Hint: As practice at using the Python Imaging Library, try writing a small program to print the individual pixel values in an image, as we've done above.)

Watermarking

The examples above show that a greyscale image requires only about a third the storage space of a full-colour one of the same dimensions. This makes it possible to 'hide' a secret greyscale image inside a colour image, in a way that is barely detectable to the naked eye.

Our goal is to hide a secret greyscale 'watermark' inside a colour image by changing the channel values associated with certain pixels. So that the watermark is not visible to the naked eye, we want to do this by making only a tiny change to each channel value. Since each pixel in our greyscale watermark is represented by a single value containing up to three decimal digits, and each pixel in our coloured image is represented by three channel values, red, green and blue, our approach is to hide one digit of the watermark's pixel value in each coloured pixel's channel.

For instance, assume that we want to hide greyscale pixel value

209

in a coloured pixel consisting of the following red-green-blue channel values.

(103, 142, 52)

To do this we will replace the least-significant digits of the channel values with each of the three digits from the greyscale value. The result in this case would be:

(102, 140, 59).

In most cases these minor changes to the coloured image's pixels will be invisible to the naked eye, but will still allow us to retrieve the watermark later.

To get the original greyscale brightness value back from the watermarked pixel, we merely need to reconstruct the original three-digit value from the least significant digits of the red-green-blue channel values.

Demonstration

As a concrete example of the process, four demonstration files have been enclosed with these instructions. You can experiment with these files to help you develop your functions.

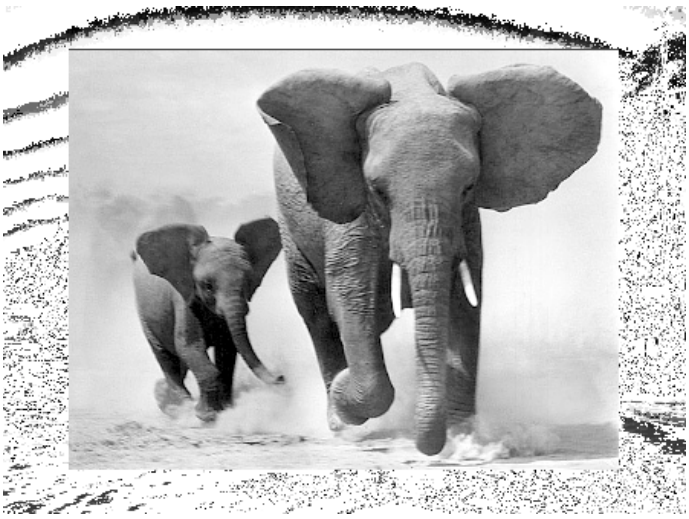
1. `Copenhagen.bmp` is a normal bitmapped colour image.
2. `elephants_watermark.bmp` is a greyscale watermark image that we want to hide in the original colour image.
3. `Copenhagen_X.bmp` is the original image with the watermark embedded in the centre. If you look closely you can see some variations in the blue sky in the photograph caused by the changes we made to the colour pixel values, but these effects are not noticeable at all in less uniform parts of the image such as the houses. This is the file that your `add_watermark` function should produce when given image `Copenhagen.bmp` and watermark `elephants_watermark.bmp`. For convenience, we use a '`_X`' suffix to identify files with embedded watermarks (although in practice we normally wouldn't want to draw attention to the fact that the file contains a hidden watermark!).
4. `Copenhagen_X_O.bmp` is the watermarked image processed so that the watermark is revealed. This is the file that your `reveal_watermark` function should produce when given watermarked image `Copenhagen_X.bmp`. For convenience, we use a further '`_O`' suffix to distinguish files whose embedded watermark has been revealed.

A copy of all four files appears below. (The images on the following pages are not to scale.) Notice that the watermark must be placed in the centre of the original image. You can assume that the greyscale watermark will be no larger than the coloured image it is to be embedded in. In this case `Copenhagen.bmp` has dimensions 447×333 pixels and `elephants_watermark.bmp` has dimensions 360×276. Also notice that when the watermark is revealed, the surrounding pixels from the original image produce random grey blobs because no greyscale data is encoded in them.

`Copenhagen.bmp` (the original image)

`elephants_watermark.bmp` (the secret watermark)

`Copenhagen_X.bmp` (the watermarked image)



`Copenhagen_X_O.bmp` (the watermarked image processed to reveal the hidden watermark)

The Python Imaging Library

To complete this task you first need to install the Python Imaging Library. This is a Python module that contains a large number of functions for manipulating images. Documentation describing the library can be found at:

http://www.pythonware.com/library/pil/handbook/index.htm.

The library itself can be downloaded from:

http://www.pythonware.com/products/pil/.

Make sure that you choose the appropriate package for Python Version 2.7. (Mac OS X users who have a 32-bit implementation of Python should install disk image `PIL-1.1.7-python.org-32bit-py2.7-macosx10.3.dmg`.)

Consult the documentation to learn how to use the library. Briefly, the main steps for manipulating whole images from files are as follows. Firstly, you need to import the `PIL` module into your Python program. You can then open an image file by creating a 'handle' to access it, e.g.:

```
image = Image.open(filename)
```

Once the file is open you can extract all the pixels from the image into a list as follows:

```
pixels = list(image.getdata())
```

For greyscale images each pixel is represented by a single integer value as explained above. For coloured images each pixel is represented by a Python 'tuple' of three red-green-blue values, which is represented simply as three comma-separated numbers in round brackets as shown above. You can use indexing to access individual tuple values just like Python lists. (See the Python documentation to learn more about the tuple data type, if necessary.) Once you have made the changes you want to the list of pixels, they can be written back into the image as follows,

```
image.putdata(pixels)
```

and the image can be saved to a new file in the current folder:

```
image.save(newfilename)
```

Robustness

Since your functions must manipulate external image files, there is a risk that your code will not work correctly because the necessary input files do not exist or the required output file cannot be created. Rather than 'crashing' with unhandled exceptions, your functions must execute to completion despite this possibility. Each of your functions must return a Boolean result, with **`True`** indicating that the files could be accessed successfully and **`False`** indicating that there was a problem accessing the files.

Specific requirements

To complete this task you must develop two Python functions which successfully perform the watermarking functions described above. Use the provided question file `image_watermarking_Q.py` as your starting point, with the '`_Q`' suffix removed. This file contains a number of doctests that run your functions on the supplied image files. (These tests **don't** confirm that you have correctly manipulated the images, however!)

- Function `reveal_watermark` must accept the name of a watermarked image file, e.g., `BryceCanyon_X.bmp`, and create a new image file, e.g., `BryceCanyon_X_O.bmp`, that reveals the hidden watermark. Note that the watermarked image file will be distinguished by a '`_X`' suffix on its filename, and the new file to be created should have an additional '`_O`' suffix added. The function must return **`True`** if it was able to access the files correctly, **`False`** otherwise. To reveal the hidden watermark this function must replace each red-green-blue tuple with a single greyscale value, as explained above.
- Function `add_watermark` must take two parameters, a colour image file name, e.g., `Flowers.bmp`, and a greyscale watermark file name, e.g., `dog_watermark.bmp`, and it must create a new image file, e.g., `Flowers_X.bmp`, with the greyscale watermark hidden in the colour image. Note that the watermarked file must be distinguished by a '`_X`' suffix on its filename. The function must return **`True`** if it was able to access all files correctly, **`False`** otherwise.

  The `add_watermark` function must embed the watermark **in the centre** of the image. (In one of the supplied tests the two images are the same size, to help you develop your code incrementally.) The watermark is encoded by hiding each of the greyscale value's digits in the coloured channels' values as described above.

We have supplied a number of image files which are used by the tests. Those with '`_X`' suffixes already contain watermarks and are used to test your `reveal_watermark` function. Files without '`_X`' suffixes are used to test your `add_watermark` function by first adding a watermark and then revealing it again.

Development hints

- The `reveal_watermark` function is much simpler than the `add_watermark` one, so you should implement `reveal_watermark` first. In particular, it's difficult to know if your `add_watermark` function is working if you can't reveal the result!
- Above we explained how to use the Python Imaging Library to extract and modify *all* the pixels from an image file, but it's also possible to manipulate just a particular rectangular *region* of an image, which can be very helpful for the `add_watermark` function when trying to position the watermark image in the centre of the main one. Consult the *PIL Handbook* documentation for examples of extracting particular regions of an image using the `crop` function.
- In your `reveal_watermark` function you will need to create a new image variable to hold the greyscale pixels. Since colour is the default option for images in PIL you will need to specify `Image.new("L", size)` when creating the image variable. The `"L"` option says that the image is intended to be 8-bit greyscale, not colour.
- One of the tricky parts of this task is knowing how to manipulate the digits of the numbers as described above and you should make sure you know how to do this first. Recall from our first lecture that given two integers `x` and `y` in Python then '`x / y`' is the whole number result of dividing `x` by `y` and '`x % y`' is the remainder of dividing `x` by `y`. With respect to the example above, therefore, note that we can extract the necessary three digits from the greyscale pixel value 209 as follows:

  `(209 / 100) % 10` returns `2`

  `(209 / 10) % 10` returns `0`

  `(209 / 1) % 10` returns `9` (as does just `209 % 10`, of course)

  Also note that `(x / 10) * 10` returns the value of `x` with its least-significant digit changed to zero. Thus the first of the greyscale digits, 2, can be hidden in the corresponding red channel value, 103, as follows:

  `(103 / 10) * 10 + 2` returns `102`

  Similarly for the other two colour channels.

  However, keep in mind that valid pixel values must always be in the range 0 to 255, inclusive. What happens, therefore, if adding the greyscale digit at the end produces a number greater than 255? For example, adding greyscale digit 8 to the end of colour channel value 253 will produce an illegal brightness value of 258. (Typically this will show up as an obvious black pixel in most picture viewers.) In this case you should make the *smallest possible* adjustment to the channel value to ensure that it's in the valid range, while still retaining the greyscale digit at the end. We'll leave it to you to figure out what this adjustment is.

- Getting a three-digit greyscale value out of a coloured pixel is even easier because `x % 10` returns the value of the least-significant digit in `x`. In the case of the watermarked colour pixel value (102, 140, 59) observe that

```
((102 % 10) * 100) +
((140 % 10) * 10) +
((59 % 10) * 1)
```

returns 209.

Deliverables

The deliverables for this task is as follows.

1. A complete Python program `image_watermarking.py` that achieves the desired functionality. This should be developed from the 'question' template file `image_watermarking_Q.py` but with the '`_Q`' suffix removed.
2. A completed "statement" at the beginning of the Python file to confirm that this is your own work and to identify your programming pair. (We can't award you marks if we don't know who you are!) Also you must indicate the percentage contribution made to the whole portfolio by both students. We do *not* expect a precise 50/50 split; an accurate and honest assessment is appreciated. This percentage will not affect your marks except in cases of *extreme* imbalances or disagreements.

Apart from working correctly your program code must be well-presented and easy to understand, thanks to (sparse) commenting that explains the *purpose* of significant code segments and *helpful* choices of variable and function names.

For our convenience, please do **not** include any image files in your submission via Blackboard because these take up a lot of space given that we have such a large number of students. We will supply the images needed to test your program during marking.