Python Practice 6: Image Processing

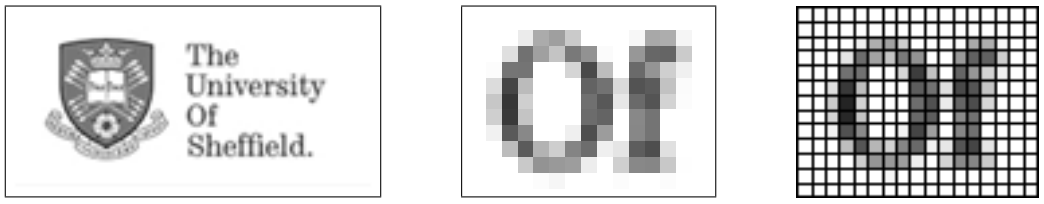
We have seen how lists and arrays can be used to hold *sequences* of values, and how we can use *loops* to operate over them. In this lab, we look at using *multidimensional* arrays of values, and the use of *nested loops* to operate over them. More specifically, we use multidimensional arrays to represent *images*, and use nested loops to perform image processing operations.

Download the lab files: Start by downloading the lab class files and storing them in a sensible location on your U drive. These are the two image files: `chick.png` and `che.png`.

1 Images represented as numeric arrays

1.1 Black-and-white images as 2D arrays

A 'black-and-white' or *greyscale* image, such as that below left, consists of a 2D *grid* of **pixels**, or *basic image units*. This image consists of a grid of 65×134 such pixels. If we expand the image area with the word "Of", as below centre, we can see the pixels as separate patches of light or dark, and this is clearer still if we impose a grid of lines between the pixels:



For greyscale images, each pixel can be represented using a single *intensity* value, e.g. we can use a scale 0.0 to 1.0, with a value 0.0 giving a fully black pixel, and 1.0 a fully white one. The intensity values of a greyscale image can be very naturally be stored within a 2D numeric array of the kind provided by the `pylab` module (as discussed in the last lecture).

1.2 Colour images as 3D arrays

Colour images are also 2D grids of pixels, but require *colour* information for each pixel. In the RGB approach to representing images, each pixel has separate intensity values for the *primary* colours *red*, *green* and *blue*. (See `http://en.wikipedia.org/wiki/RGB_color_model`.) The three primary colours combine to give any other colour. This data is naturally stored in a 3D array of size $d1 \times d2 \times 3$, where $d1/d2$ are the image pixel dimensions, and where the lowest level triple stores the RGB values of a single pixel.

`pylab` can read image files, and load the intensity values into arrays of precisely this form. Create a code file for this exercise, and in that file, enter the following code to load and display the image `chick.png` (for which the image file must be in the same folder as your code file):

```
from pylab import *
img = imread('chick.png')
imshow(img)
show()
```

The command `imread` reads the image, and returns a 3D numeric array of its intensity values, which is assigned to the variable `img`. Having run the code, you can check the value of `img` in the interpreter. If you type `img` at the interpreter prompt, it will print something of the variable's value — although it is too big to print fully. We can get the dimensions of the array by checking its "`.shape`" attribute, and can assign the three values to a triple of variables, as in the following:

```
>>> img.shape
(100, 75, 3)
>>> (rows, cols, d3) = img.shape
>>> cols
75
```

As shown below, we can use a triple of indices to access a single intensity value in the array, or we can use just *only two* indices to reach in to access the *triple* of RGB values for a *single pixel.*

```
>>> img[0, 0, 0]
0.94901961
>>> img[0, 0]
array([ 0.94901961,  0.84313726,  0.67058825], dtype=float32)
```

There are two alternative ways that we can address this representation to modify the image:

Method 1 (addressing individual intensities):
First, we can use a *three-level* nested loop, giving access to *individual* intensity values in the 3D array, as in the following example, which 'flips' low intensities to high ones, and vice versa, producing a rather ghostly effect (try it yourself, to see the effect):

```
(rows, cols, d3) = img.shape
for i in range(rows):
    for j in range(cols):
        for k in range(d3):
            img[i, j, k] = 1 - img[i, j, k] # DO SOMETHING TO INDIVIDUAL INTENSITY VALUE
```

Method 2 (addressing pixel triples):
Alternatively, we can use a *two-level* nested loop, to give us access to the value triples of *pixels*, and make a change that affects the entire pixel *in one step*, as in the following example:

```
(rows, cols, d3) = img.shape
for i in range(rows):
    for j in range(cols):
        pixel = img[i, j]
        if sum(pixel) < 1.5:
            img[i, j] = (.0, .0, .0) # DO SOMETHING TO CHANGE PIXEL VALUES IN ONE STEP
```

In this example, "`img[i, j]`" reaches in through two levels of the 3D array to access the *triple* of values of the *pixel* at grid position `(i, j)`. The conditional checks if the three values *sum* to less than 1.5 (in other words, whether their *average* is less than 0.5 — the *middle* intensity value). If so, the three values of the pixel are over-written with a *triple of zeros*, in a single assignment step, i.e. the pixel is changed to be black. Again, try this out for yourself.

2 Che, Cool Che, and Funky Che

We have seen enough for you to be able to write your own code performing some image manipulations. Working in a new code file, write some code to load and display the image `che.png`, which is based on the earlier example for the *chick* image. This image is the original black-and-white photograph of the famous Argentinian revolutionary *Che Guevara* from which the ubiquitous *pop-art* images are derived.

Although the image is greyscale, you'll find that it is stored just like the chick image, i.e. with pixels encoded as triples of RGB values (to see this, check its `shape` attribute). For greyscale

images stored this way, however, the three components have *the same value*, i.e. as when no colour dominates, the effect produced is in *shades of grey*. Thus, a triple `(1.0, 1.0, 1.0)` produces a white pixel, `(0.0, 0.0, 0.0)` a black pixel, and `(0.5, 0.5, 0.5)` a mid-grey pixel.

Task: Extend your code so that it modifies the image (or a copy of it) to produce the various image effects described below. (**Note:** You can copy an array using the `array` function, e.g. as in "`img1 = array(img)`", so you can then modify `img1` without altering `img`.)

Each image effect requires you to write some code with nested loops, to traverse the image array and modify intensity values according to various rules, as exemplified in the previous section. Some specific suggestions for image effects follow, but you can think up some more of your own. In each case, consider whether the task is best handled as an instance of `Method 1` above, the simpler method that addresses/modifies only individual intensity values, or if the more complicated `Method 2` is needed, to evaluate and replace entire pixels in one step.

1. The classic *Che* poster image has a *strongly monochrome* effect, i.e. dividing into areas that are solid black and solid white (or with some other colour in place of white). You can achieve this effect by applying the following simply rule to individual intensity values (Method 1): all values below 0.5 are modified 0.0, and all others to 1.0. Try this out.

2. As a variant of the previous case, change the image to be solidly black and *red*. This idea is most naturally implemented by a method working at the level of pixel triples (Method 2). You can use `(1.0, 0.0, 0.0)` for a strongly red pixel.

3. As another variant of the above, produce a version with black areas as above, solid white for Che's face, and all other areas as red — again, a classic version of the poster. **Hint:** The numeric scales on the left and bottom of the image can be used to identify row and column values that determine a rectangle that bounds the area of Che's face. These values can then be used to implement having different behavour within/without this area (e.g. whether red/white is assigned for non-dark pixels).

4. For a rather more funky effect, you could try the following: change any pixel with intensity value above 0.66 to be strongly red, any below 0.33 strongly blue, and the remainder green. Again, this is most easily handled as an instance of Method 2.

5. For a slightly more challenging task, copy the chick image to *overlay* the Che image. Since the chick image is *smaller* than the Che image, we can simply traverse the chick image and assign each of its pixels to the corresponding location in the Che image. Where in the larger picture does the chick appear, and why? Next consider how to make the chick appear elsewhere on the Che image, e.g. can you make it appear where Che's face should be?