Image processing lab session

In this session we're going to be looking at how bitmapped graphics can be processed using some common image manipulation techniques.

Software requirements

You will need Python v3.5 or above and the following Python libraries:

- Imageio - a Python library that provides an easy interface to read and write a wide range of image data. You can read the docs here.
- Numpy - a Python library is a general-purpose array-processing Python library which provides handy methods/functions for working with multidimensional arrays. You can read the docs here.

Ada Computer Science

This theory content covering bitmapped graphics on Ada CS can be found here.

Worksheet 1: Images processing basics

The aim of this session is to get acquainted with some basic image processing techniques.

To work with images in Python, you'll need to be able to read them in from a file. Fortunately this is very straightforward using the **imageio** library.

Exercise 1.1 - Opening a file

1. Make a folder for your files.
2. Get the file 'balloon-100.jpg' (for other images see Appendix) and save it **in the same folder**.
3. Next, create a new Python file and type the following code. Save it **in the same folder**.

```python
import imageio

def main():
  # img1 - existing image
  infile = "balloon-100.jpg"
  img1 = imageio.imread(infile)
```

The first line imports the **`imageio`** library. It is conventional to put your import statements at the top of the program file so that anyone looking at your code is aware of any dependencies.

You do not have to have a **`main`** function in Python, but this name is used in many other programming languages and is a convention used by the Ada CS team.

To run your code you need to call your **`main`** function.

4. Add the following lines of code at the **end** of your file.

```python
if __name__ == "__main__":
  main()
```

This is another Python coding convention. Fundamentally, it specifies that if this program file is run directly, the function named **`main`** will be called automatically. Conversely, if the program file is imported into another program file, the function named **`main`** will not be called.

5. Run your code. If you don't get any error messages you have successfully opened and read the file. Otherwise check your code carefully and correct any errors.

Exercise 1.2 - Displaying image properties

Your image is now available as an **array of pixels** and you can display information about it.

6. Create a new function in your program file. Type the following code above the **`main`** function (so it appears as the first function in the program file).

```python
def display_image_properties(img):
  '''Display some information about the image.'''
  print(f"Type: {type(img)}")
  print(f"Shape: {img.shape}")
  print(f"Data type: {img.dtype}")
  print("\n")
```

This function uses **f-strings**. The value in the curly braces is substituted into the string in the position specified.

The final print statement displays a blank line.

7. Now you need to call the new function. Add the following line of code to the end of the **`main`** function.

```python
display_image_properties(img1)
```

8. Run your code. It should show you that you've got a **numpy.ndarray**. Its dimensions are 82 by 100 by 3 and each element is of type **uint8** (an 8-bit unsigned integer).

```
Type: <class 'numpy.ndarray'>
Shape: (82, 100, 3)
Data type: uint8
```

Exercise 1.3 - Displaying pixel RGB values

Bitmapped graphics are made up of pixels (a pixel is a picture element). When you read the image file using the **`imageio`** library you will have an array of pixels. The array has three dimensions.

The first dimension selects a row and the second a column within that row. In this way you can select a specific pixel; for example, **`img[0][0]`** will reference the pixel in the top left corner. This image is 82 pixels high by 100 wide, so the bottom right pixel is referenced as **`img[81][99]`**

The third dimension is the colour. This contains three values corresponding to the amount of red, green and blue the pixel contains.

9. Create another new function in your program file. Type the following code above the **`main`** function (you should always have your main function as the last function in the file).

Python
```python
def display_some_pixel_data(img):
  '''Display some information about specific pixels.'''
  # RGB values of the top left pixel.
  print(f"RGB:", img[0][0])
  # Red value of top left pixel.
  print(f"R: {img[0][0][RED]}")
  # Green value of top left pixel.
  print(f"G: {img[0][0][GREEN]}")
  # Blue value of the top left pixel.
  print(f"B: {img[0][0][BLUE]}")
  print("\n")
```

This function introduces some **constants** to hold the index values to reference the position of the red, green and blue pixel data. This will make your code easier to read and is also good practice.

The Python programming language does not actually have constants but we can use the convention of specifying global variables with upper case names so that a programmer is aware that these values should not be changed.

10. Add the following lines of code at the top of your program file (below the import statement):

Python
```python
RED = 0
GREEN = 1
BLUE = 2
```

11. Now you need to update `main` to call your new function. Add the following line of code to the end of the main function.

```python
display_some_pixel_data(img)
```

12. When you run your program you should see this additional output:

```
RGB: [189 214 254]
R: 189
G: 214
B: 254
```

The array has three dimensions; the third dimension is the colour. This contains three values corresponding to the amount of red, green and blue the pixel contains. This is the pixel RGB value. For a specific pixel the first value (index 0) is the amount of red, the second (index 1) the amount of green, and the third (index 2) the amount of blue.

Each colour value is an 8-bit number (i.e. it takes a value in the range 0–255), with 0 being none of that colour and 255 being as much of it as possible. Therefore an RGB value of (0, 0, 0) is black, (255, 255, 255) is white, (255, 0, 0) is red, (0, 255, 0) is green and (255, 255, 0) is yellow, etc.

Some images have a fourth colour. This is called an alpha channel and represents the amount of transparency the pixel has (how much you can see whatever is underneath it). An alpha value of 255 means the pixel is totally opaque (nothing below can be seen) and 0 means it is fully transparent (you only see what is underneath and nothing of the pixel itself). We say these pixels have RGBA values.

Exercise 1.4 - Creating a new image

Now you will add some code to create a new image. You will use the **numpy** library to make an array to represent an image of a specific size (in this case 100 by 200 pixels, each having 3 colour values - RGB) by specifying the shape you want in a call to `np.zeros`. This will create a numpy array of zero values, so every pixel will be black!

13. Add this code to the end of your **`main`** function

```python
  # img2 - new image
  img2 = np.zeros((100, 200, 3), dtype=np.uint8)
  display_image_properties(img2)
```

14. Before you can run your program, you need to add code to import the **numpy** library. Add the following line to the **top** of your program file below the existing import statement.

```python
import numpy as np
```

15. When you run your program you should see this additional output:

```
Type: <class 'numpy.ndarray'>
Shape: (100, 200, 3)
Data type: uint8
```

Exercise 1.5 - Creating a new image the same shape as another

You can also create an image that is exactly the same shape as another image using its shape.

16. Type the following code as new lines at the end of your **`main`** function

```python
  # img3 - new image same shape as img2
  img3 = np.zeros(img2.shape, img2.dtype)
```

```
display_image_properties(img3)
```

17. Run your program and you should see a duplicate set of information for the new file because it has copied the dimensions of img2 using img2.shape.

Exercise 1.6 - Turning pixels blue

Pixel colour values can be manipulated just like other variables. We can, for example, make all pixels in our new image blue by setting the corresponding RGB values.

18. Create another new function by typing the following code above the main function.

```python
def turn_pixels_blue(img):
  '''Change colour of all pixels to blue.'''
  for row in range(img.shape[0]):
    for col in range(img.shape[1]):
      img[row][col][RED] = 0
      img[row][col][GREEN] = 0
      img[row][col][BLUE] = 255
```

This function uses nested for loops to iterate through each row (first dimension of the array) and then each column (second dimension of the array) within the row to process every pixel. The pixel's red and green values are set to 0 and the blue value is set to 255 (max setting).

19. Now add a line to the end of **`main`** to call the new function, and another line of code to write the updated pixel data to a new file .

```python
turn_pixels_blue(img3)
```

```
imageio.imwrite("blue_image.jpg", img3)
```

You can now open the file (it will be in the same folder as your other files for this project) and check that it is indeed blue!

Exercise 1.7 - Creating a square of yellow pixels

Now you can try to make a yellow square in the centre of the image, by changing the corresponding pixel values.

20. Create another new function by typing the following code above the main function (so that the new function appears as the fourth function in the program file).

```python
def create_yellow_square(img):
  '''Create a yellow square within image.'''
for row in range(40, 60):
    for col in range(90, 110):
      img[row][col][RED] = 255
      img[row][col][GREEN] = 255
      img[row][col][BLUE] = 0
```

21. Add a couple of lines to the bottom of your main function to call the new function and write the updated array data to a file.

```python
create_yellow_square(img3)
imageio.imwrite("yellow_square.jpg", img3)
```

You can now open the new file (it will be in the same folder as your other files for this project) and check that your code has worked as expected.

Notes:

The `create_yellow_square` function is not **generalised** to work with any image. The square will always start at 40,90 and end at 60,110 (as these values are hard-coded). If the size of the image you were working with was smaller, your program might "crash" because there may not be 110 columns. It would be better if the centre position was calculated from the shape of the image.

The same is true for the function `display_some_pixel_data`. This can be generalised by specifying additional parameters so that you can pass in the coordinates of the pixel you want to access.

In general you should avoid hard-coded values in your functions. Use constants, additional parameters and calculated values to generalise your code.

And another thing .....

Because the Python image is a **numpy** array, you can use conventional Python indexing or the **numpy** convention. Thus:

```
img[23][46][6]
```

can be written as

```
img[23,46,6]
```

`We will use this convention in our code examples for worksheets 2-4.`

Worksheet 2: Colour transformations

Now that you have learnt the basics of how digital images are represented and manipulated, you are ready to tackle a couple of interesting transformations we can do with colour:

- convert to sepia
- convert to grayscale

We'll work through the sepia transformation step by step.

Exercise 2.1 - Sepia transformation

Sepia colour makes images look like those seen in old photographs. To convert a picture to sepia requires adjusting the RGB values of each pixel. To do that, we use the following formulae:

```
red = img[row, col, 0]
green = img[row, col, 1]
blue = img[row, col, 2]
new_r_value = min((0.393 * red ) + (0.769 * green ) + (0.189 * blue), 255)
new_g_value = min((0.349 * red ) + (0.686 * green ) + (0.168 * blue), 255)
new_b_value = min((0.272 *red ) + (0.534 * green ) + (0.131 * blue), 255)
```

To make the transformation, we start by extracting the original red, green and blue values of the pixel. Then we calculate the new sepia values for each of the pixel's RGB data. We do this by mixing the original pixel colour values together:

- For the new red value, we take 0.393 of the original red, 0.769 of the original green and 0.189 of the original blue to create the new value.
- For the new green value, we take a different mixture of the old values (0.349 red, 0.686 green and 0.168 blue)
- For the new blue value it's different again (0.272 red, 0.534 green and 0.131 blue).

Recall that each colour code is a value between 0 and 255. These calculations may result in a value higher than 255. To combat this, you must **cap** the value of any colour code at 255 using the **min** function.

`min(calculated_value, 255)`.

You will also need to **round** the numbers to integer values.

Now you have seen the formulae behind the sepia transformation, you can write code to apply the formula to an image.

1. Open your Python file and create a new function with a meaningful name.

```python
def apply_sepia_transformation(img):

  '''Create and return a sepia version of an image'''
```

2. Add a line of code to **create a new numpy array** to hold your transformed image. The array must be the same shape as the original image (look back to how you did this previously).
3. Now add code to iterate over all pixels in the original image (loop through every row and every column in that row). Again, you have already written code to do this in a previous function.
4. Add code to extract the colour codes for each pixel and implement the formulae for a sepia transformation. Don't forget to round your new (calculated) colour codes, and make sure they are no greater than 255.
5. Apply the new colour codes to the relevant pixel in the new array.
6. Finally add code to write your updated array as a new image file.
7. Add a line of code to your **main** function to call your new function. Pass in the reference to an image to transform.
8. Run your program. Test that it has worked correctly by opening and viewing the new image file.

Exercise 2.2 - Greyscale transformation

Greyscale images don't have RGB values for each pixel, but just a single greyscale value in the range 0–255. The following formula calculates a new greyscale value for each pixel in a colour image.

```
red = img[row, col, 0]
green = img[row, col, 1]
blue = img[row, col, 2]
new_greyscale_value = min((0.3 * red) + (0.59 * green) + (0.11 *  blue, 255)
```

We don't calculate a simple average of the three RGB values because the human eye sees each colour differently. You can experiment with the weightings if you like.

9. Create a new function (with a meaningful name.) to convert an image to grayscale.
10. Add a line of code to **create a new numpy array** to hold your transformed image. The array must be the same shape as the original image (look back to how you did this previously).
11. Iterate through all of the pixels and apply your formula. .Remember to write the image to a file so you can check your work.

Worksheet 3: Point processing

Exercise 3.1 - Brightness

You can increase brightness by raising each pixel's RGB values, and lower brightness by reducing them. The following formula is a very simple way to do this:

new_value = original_value + brightness_factor

A factor of > 1 will brighten the image; < 1 will make it darker. Remember that you need to keep each colour value in the range 0 - 255.

1. Create a new function (with a meaningful name) to adjust the brightness of an image. Specify two parameters: the image to adjust, and the brightness factor.
2. Write code to adjust each pixel's value by the specified amount.
3. Remember to call your new function from main and also write the image out to a file so you can check your work.
4. Test your code using a factor of **+50**
5. Decrease the brightness using an adjustment of **-50**
6. For a large adjustment (say **+100**), what happens if you increase the brightness and then decrease it again? What has happened?

Exercise 3.2 - Contrast

The contrast is the difference between the lightest point and the darkest point in the image. The following formula uses a contrast factor of between -1.0 and +1.0

new_value = (contrast_factor * (old_value - s)) + s

.Again, you need to make sure you keep each colour value in the range 0 - 255.

7. Increase the contrast of a photo using s = 127 and a factor of 0.3
8. Try different values of a and s to see how contrast is affected.

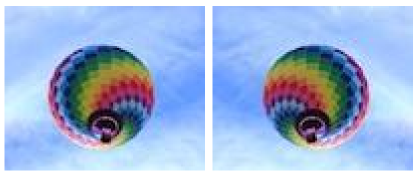
Worksheet 4: Affine transformations

Having looked at colour transformations in worksheet 2, we can now consider **affine transformations**. These change properties like the scale or rotation of an image. There is a Wikipedia page on digital image processing that shows examples of such transformations.

Affine transformations don't always change the actual RGB values of each pixel (although they can do), but do change their positions within the image. In other words, we create a new image array and copy pixel values from the original image, placing them in different locations. You may also need to alter the dimensions of the transformed image when it is created if you rotate by anything other than 180° or scale it.

Exercise 4.1 - Reflection

You can create a mirror image by reflecting about a line through the centre of the image. The first image below has been reflected vertically to produce the second image:



To obtain a vertical reflection you can use this formula:

new_col = width – col –1

For example, if the image is 100 x 100, the pixel which is at (80, 15) will be moved to (80, 84)

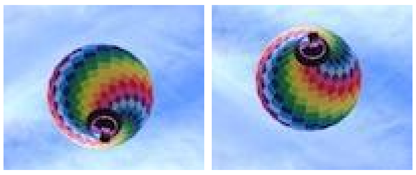
- The row value remains the same
- The new col value is 100 - 15 - 1 = 84

This keeps the pixel in the same row but changes its column.

1. Open your Python file and create a new function with a meaningful name.
2. **Add a line of code to create a new numpy array to hold your transformed image.** The array must be the same shape as the original image.
3. Loop through the rows and columns of the original image and calculate the new_col value

4. Copy the pixel to the new coordinates in the new array.
5. Remember to write your array to a file so you can check your work.

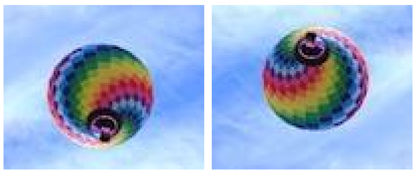
To reflect the image horizontally, you would need to keep the column index the same but calculate and use a new row index. Can you do this?



Exercise 4.2 - Rotation through 180°

Rotation also involves copying pixel values to new locations. Let's start by rotating through **180°** (so we can keep the same image dimensions).

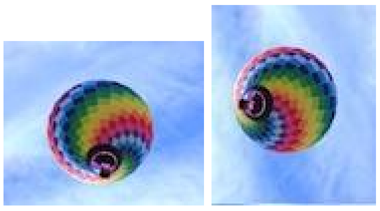
To perform the rotation you need to move each pixel to new row **and** column coordinates.



You need a combination of the lines you wrote for the horizontal and vertical reflections.

6. Open your Python file and create a new function with a meaningful name.
7. **Add a line of code to create a new numpy array to hold your transformed image**. The array must be the same shape as the original image.
8. Loop through the rows and columns of the original image and calculate the new_col value and the new_row value.
9. Copy the pixel to the new coordinates in the new array.
10. Remember to write your array to a file so you can check your work.

Exercise 4.3 - Rotation through 90°



Study the formula in the snippet below. You may like to do some manual calculations to ensure that the new positions are correct.

```python
# For each pixel, calculate its new co-ordinates (90 rotation)
new_row = col
new_col = new_width - row - 1
# write pixel to new position in the new (rotated) image
rotated_img[new_row][new_col] = img[row][col]
```

11. Open your Python file and create a new function with a meaningful name.
12. Add a line of code to create a new numpy array to hold your transformed image. **If your image is not square, you must transpose its dimensions for the new numpy array**. A 100 x 80 image will be rotated to create an 80 x 100 image.
13. Loop through the rows and columns of the original image and calculate the new_col value.
14. Copy the pixel to the new coordinates in the new array.
15. Remember to write your array to a file so you can check your work.

Exercise 4.4 - Scale

Scaling an image not only requires copying pixel values to new locations, it also involves calculating new RGB values. When the image size is increased, you need to define new RGB values for the new pixels. When the size of an image is decreased, you need to combine RGB values to account for the loss of pixels. This is why resizing a bitmap graphic will always cause a lack of fidelity (and the change will be significant if the amount of resizing is significant or the operation is done repeatedly).

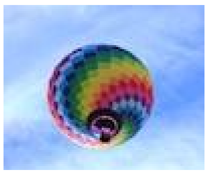
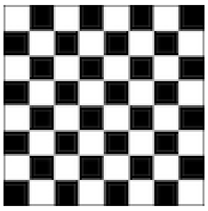
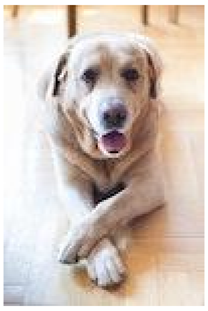
Let's have a go at reducing the size of an image using a very simple technique:

1. Create a new numpy array that is 50% smaller in both dimensions than the original. So, if the original image is 100 x 82 , the new image will be 50 x 41.
2. Read in a block of 4 pixels (2 rows and 2 columns) from the original image and average the RGB values from these pixels to create a new single RGB code.
3. Write this new RGB code to the new array

This is a very crude way of calculating the new RGB value. Can you think of a better method? You can read more about image resizing here.

Appendix 1 - Sample images

The following images are provided for you to test your transformations. Each image is provided as 100px, 300px or 600px in its largest dimension. Using the smallest version of an image will be significantly faster than if you use the larger versions. Click the required resolution in the table to link to the downloadable image file. All images are in the public domain (so you are free to use them without licence or attribution).

Image	Links
	balloon-100.jpg balloon-300.jpg balloon-600.jpg
	chessboard-100.png chessboard-300.png chessboard-600.png
	coffee-100.png coffee-300.png coffee-600.png
	labrador-100.jpg labrador-300.jpg labrador-600.jpg
	lake-100.jpg lake-100.jpg lake-100.jpg

Acknowledgements

These worksheets and code snippets are based on materials produced by, and used with permission from, the University of Cambridge.