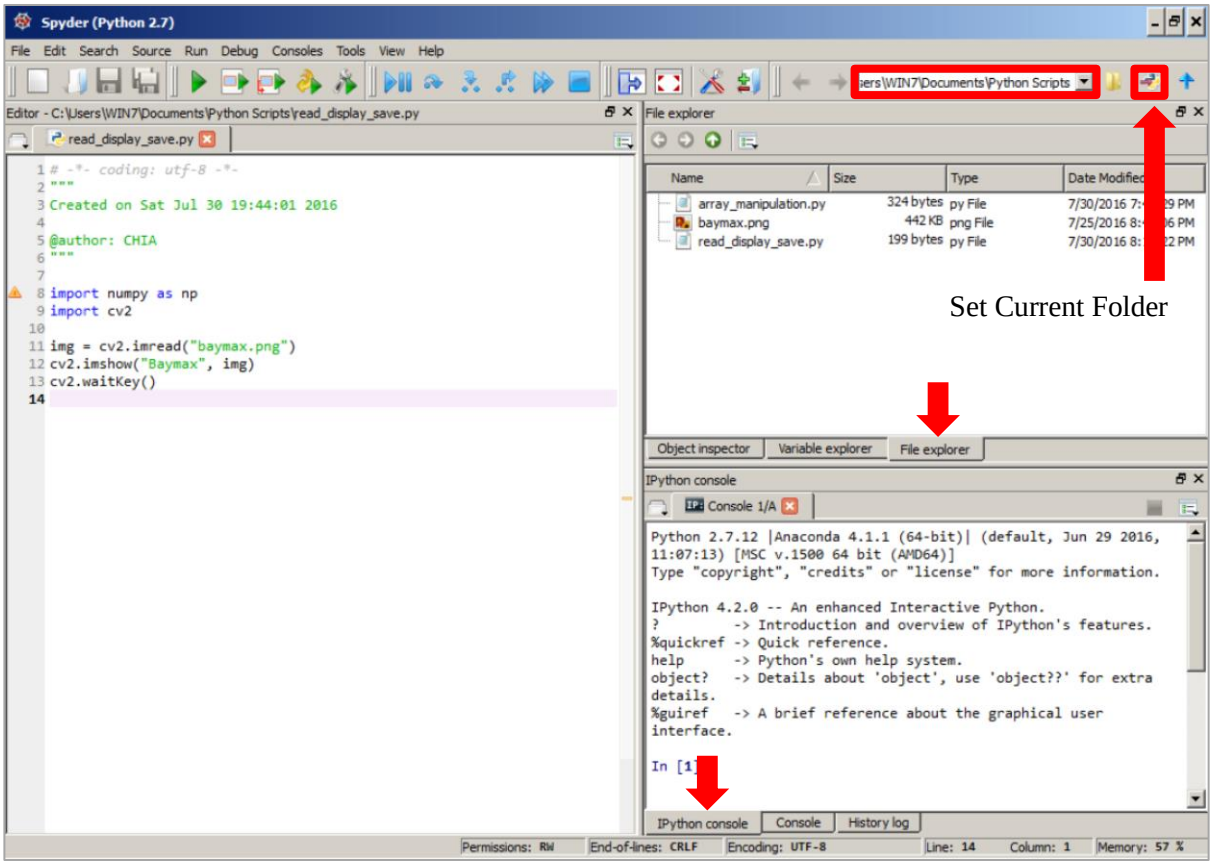
CSC2014: Lab 3 – Point Operation

A. Read, Display and Save

Before we start to read and display an image, we should first put the image into your working folder. You may check where is your current working folder by referring to the path shown in the top right corner (as shown below) of Spyder.



You can stick to the existing folder or change to the one you preferred. Remember to click the **Set Current Folder** button if you are changing to a new one. This reduces the hassle of having to specify the exact path when reading an image. Use the File Explorer to see if you are in the correct working folder. Then create a new script file, enter the following codes, and run the file.

```
import numpy as np                      #Import NumPy
import cv2                              #Import OpevCV
img = cv2.imread("baymax.png",1)        #Read and save the image into img
cv2.imshow("Baymax", img)               #Use default viewer to display img

#Optional
cv2.waitKey()                           #Wait for user to press any key
cv2.destroyAllWindows()                 #Close the window
```

Important
In certain situations, you might want to specify the complete path. For example, if you put the images in C:\Python Scripts, you will have to add this to the front of your image filename. The prefix 'r' has to be included as well or else '\' will be treated as escape character.

```
img = cv2.imread(r"C:\Python Scripts\baymax.png",1)
```

We are using the functions from OpenCV to read and display an image. The **imread** function can take two arguments, the first argument is the filename, and the second argument is the type of image (put '**1**' for colour image or '**0**' for grayscale image). If the second argument is omitted, then it will take the default value of one. In the sample codes, the image is first saved into a variable called **img**.

If the image is a colour image, **img** will have three planes (layers), each represents a channel in the RGB colour model. It is important to note that the channels are arranged in the order of Blue (B), Green (G), and Red (R). They are represented by index number **0**, **1**, and **2** respectively. If the image is a grayscale image, then **img** will only consists of a single plane.

The **imshow** function is used to display the image stored in **img** by using the default viewer of the operating system. It has two arguments, the first argument is the name given to the viewer window, and the second argument is the image to be displayed. Next, the **waitKey** function pauses the execution of the script file until you press any key on the keyboard.

Exercise 1
Read the grayscale image "**cameraman.bmp**" (download from eLearn) and change the top left quadrant of the image to black colour. Save your script file as **lab3_ex1.py**.

Assuming that we would like to save the outcome, we can make use of the **imwrite** function to write the image into a separate image file. It has two arguments, the first argument is the filename, and the second argument is the image to be written into the file. OpenCV will automatically save the image into the required format, based on the extension given in the filename (in this case, .png).

```
cv2.imwrite("new_image.png",img)          #Save img into a new image file
```

B. Image Arithmetic

There are two ways to manipulate the pixel (intensity) values, (i) use the basic Python operators **(+, -, *, /)**, (ii) use the built-in functions in OpenCV. The following codes show the use of these two methods.

```
import numpy as np                                  #Import NumPy
import cv2                                          #Import OpenCV
cman = cv2.imread("cameraman.bmp",0)                #Read the grayscale image

[nrow,ncol] = cman.shape                            #Find the size of the image
mag = np.zeros((nrow,ncol),dtype=np.uint8)          #Create a same size array
```

```
#continue on next page
#continue from previous page
mag = mag + 100                                 #Add 100 to the array

cman_add_basic = cman + 100                     #Add using basic operator
cman_add_cv = cv2.add(cman,mag)                 #Add using function
```

Challenge 1

Use the Variable Explorer, observe the difference between basic Python operator (**+**) and OpenCV function (**`cv2.add`**). What is the main difference between the two?

When the basic Python operator is used, the final output is analogous to applying a modulo function after addition. For example, 160 + 100 will result in a final output of 4 (260 modulo 256). This is different from the OpenCV function, where the pixel values are capped at 255.

It is important to note that the values are limited by the data type (**`uint8`**). In this case, the values can only range from 0-255, and the basic Python operators and the OpenCV functions will have to work around this. Which method to use is really depends on the type of output you are looking for.

However, it is not always feasible to deal with such a small range. Most of the time, it is necessary to have an array that could store values much larger than 255. The purpose is to ensure that the values will not be altered or truncated during the processing. To resolve this issue, we can either change the data type of an existing array or specify the data type when creating a new array.

But if the values are beyond the range of 0-255 after the processing, we might have a problem in showing the output, because normally a proper image (**for display purpose**) should only have values range from 0-255. Therefore, we will have to scale the values back to the range of 0-255 when it is necessary to display the output.

An example that demonstrates how to perform scaling after adding "**`cameraman.bmp`**" to "**`boat.bmp`**" is shown below.

```
cman_f64 = cman.astype(np.float64)              #Change type to float64

boat = cv2.imread("boat.bmp",0)                 #Read the grayscale image
boat_f64 = boat.astype(np.float64)              #Change type to float64

boatman = boat_f64 + cman_f64                   #Add two images
cv2.imshow("Boatman Float", boatman)            #Show the outcome
cv2.waitKey()                                   #Wait for user input

max_value = boatman.max()                       #Find the maximum value
scl_boatman = (boatman / max_value)*255         #Scale the values to 0-255
cv2.imshow("Scl. Boatman Float", scl_boatman)   #Show the outcome

scl_boatman = scl_boatman.astype(np.uint8)      #Change type to uint8
cv2.imshow("Scl. Boatman Uint8", scl_boatman)   #Show the outcome
```

Exercise 2

Given three grayscale images, "**word1.bmp**", "**word2.bmp**", and "**word3.bmp**", add them together and then display the final output. The approach you should use should be similar to the example of adding "**cameraman.bmp**" to "**boat.bmp**". At the end, you should be able to see "**Sunway University**" in your output. Save your script file as **lab3_ex2.py**.

Other commonly used arithmetic operators / functions are summarised in the table below. Please take note that element-wise multiplication / division is different from matrix multiplication / division.

Python Operator	OpenCV Function	Description
`img1 - img2`	`cv2.subtract(img1,img2)`	Image subtraction (`img1 - img2`).
`img1 * img2`	`cv2.multiply(img1,img2)`	Element-wise image multiplication.
`img1 / img2`	`cv2.divide(img1,img2)`	Element-wise image division.

Challenge 2

Given two colour images, '**dice1.png**' and '**dice2.png**', find the number of different spots between the two.

Exercise 3

Given a grayscale image, '**ufo.bmp**', adjust the contrast until you can see the hidden object in the image. Use the **imwrite** function to save the adjusted image as '**baboon.bmp**'. Save your script file as **lab3_ex3.py**.

C. Bitwise Operation

There are four fundamental logical operations, which are AND, OR, XOR, and NOT. Among them, logical-AND is probably the most frequently used operation. Generally, it is used to select a particular region of an image. In this case, one can choose to process only the chosen region instead of the entire image. To do this, we have to first create a mask that defines the region of interest.

The mask is usually represented as a binary image with only two pixel values, 0 or 255. The latter is used to indicate region that should remain after logical-AND. An example that demonstrates how logical-AND is applied to select the centre region of "**cameraman.bmp**" is shown below.

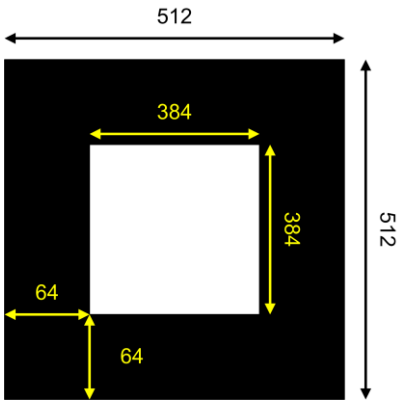
```
import numpy as np                              #Import NumPy
import cv2                                      #Import OpevCV
cman = cv2.imread("cameraman.bmp",0)            #Read the grayscale image

[nrow,ncol] = cman.shape                        #Find the size of the image
mask = np.zeros((nrow,ncol),dtype=np.uint8)     #Create a same size array
mask[64:192,64:192] = 255                       #Define region of interest

bitand_cman = cv2.bitwise_and(cman,mask)        #Bitwise logical-AND
cv2.imshow("Masked Cameraman", bitand_cman)     #Show the outcome
```

The other commonly used logical operations are summarised in the table below.

OpenCV Function	Description
`cv2.bitwise_or(img1,img2)`	Bitwise logical-OR.
`cv2.bitwise_xor(img1,img2)`	Bitwise logical-XOR.
`cv2.bitwise_not(img1)`	Bitwise logical-NOT.

Challenge 3

Create the mask as shown below and logical-AND the mask with "**`baboon.bmp`**".



-- END --