Mandatory Assignment 4

Basic Python programming (30/40 + 5 bonus points points)

University of Oslo - IN3110/IN4110

Fall 2019

Your solutions to this mandatory assignment should be placed in the directory `assignment4` in your Github repository. Your delivery should contain a `README.md` file containing information on how to run you scripts. In particular, it is important that you document how to run your tests. In addition, your code should be well commented and documented. All functions should have docstrings explaining what the function does, how, and an explanation of the parameters and return value (including types). We recommend you use a well-established docstring style such as the Google style docstrings[1]. However, you are free to choose your own docstring style. Your code should also be well formatted and readable. Coding style and documentation will be part of the point evaluation for all tasks in this assignment.

4.0 Background (0 points)

In this assignment, you will make a program for blurring an image. We will represent an image in Python as a 3-dimensional array (H, W, C), where H and W are the height and width of the image respectively, and C is the channels. For this assignment, we will represent an image by a numpy array.

To load images into numpy arrays we will use OpenCV. Both numpy and OpenCV can be installed using pip:

```
python -m pip install numpy opencv-python
```

After installing these libraries, an image stored in `filename` can be loaded to a numpy array as

```
import cv2
image = cv2.imread(filename)
```

[1] `https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html`

`image` is then a 3-dimensional numpy array with channels $C = 3$, in the order blue, green and red (BGR). Note that OpenCV uses BGR, while many other image handling libraries use the order red, green and blue (RBG). Thus, to use `image` with other image libraries (for example for plotting), you must switch the channel order. This can be done manually on the numpy array, or by the OpenCV call:

```
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

For this assignment it is not necessary to convert to RGB.

Blur can be applied to a (H, W, C) image by applying convolution with an averaging kernel. That means, for each pixel in the image, we set the corresponding pixel in the blurred image to be an average of the pixel values in the neighboring pixels. If `src` is the original image, and `dst` is the new blurred image, the application of a 3×3 averaging kernel at pixel h and w in channel c is:

```
dst[h,w,c] = (src[h,w,c] + src[h-1,w,c] + src[h+1,w,c]
              + src[h,w-1,c] + src[h,w+1,c]
              + src[h-1,w-1,c] + src[h-1,w+1,c]
              + src[h+1,w-1,c] + src[h+1,w+1,c])/9
```

Note that we treat each channel independently.

Important: There are libraries that can apply convolution kernels for you, but you are not supposed to use these. Please implement the operation manually for this assignment.

On the edges of the image, not all immediate neighboring pixels exist. The solution is to define the "pixels" outside the image to have the same value as the pixels on the edge of the image. That is, if on the left side of the image such that $h = 0$, then we define the pixel value at $[h - 1, w, c]$ to be the same as the value at $[h, w, c]$ and the pixel value at $[h - 1, w + 1, c]$ to be equal to the value at $[h, w + 1, c]$. In essence, we add a single pixel outside the image by repeating the values of the pixels at the border of the image. This process can easily be done with numpy using `pad` with *edge* mode.

OpenCV reads images as (unsigned) integers. However, after applying the averaging, the new blurred image likely contains floating point numbers. Thus, for OpenCV to understand your blurred image, you must convert the values back to integers. This can be done as follows

```
dst = dst.astype("uint8")
```

Note: Because the input image is read as an array of unsigned 8-bit integers (uint8), adding such values will cause an overflow when the sum exceeds 255. To combat such overflows, one can either divide all terms individually in the previously shown sum, or convert `src` to a type with a higher maximum value, such as unsigned 32-bit integers (uint32):

```
src = src.astype("uint32")
```

Once the blurred image is converted back to the correct type, you can save it to a file using `imwrite`.

```
cv2.imwrite("blurred_image.jpg", dst)
```

Throughout this assignment, you can use the provided image `beatles.jpg`.

4.1 Python implementation (6 points)

First, create a Python script which blurs an image using a 3×3 averaging kernel as described above. You should use Python for-loops and not vectorized operations.

You *are* allowed to use numpy for this part of the problem, but only for storing and padding the image. All computations should be done with pure python.

The computational time can easily grow very large in this task. Consider using a small image for testing during development, or resize the provided image using OpenCV. The following line of code halves the dimensions of the image.

```
image = cv2.resize(image, (0, 0), fx=0.5, fy=0.5)
```

Name of file: `blur_1.py`

4.2 Numpy implementation (6 points)

Make a similar script to 4.1 so that all computationally heavy bits use numpy arrays. Compare the runtime on some input of your choosing. How much is gained by switching to numpy? Your report should contain the parameters used to generate the images, along with the runtime for each script.

You are not supposed to use any library function that applies the convolution for you. That is, you are not allowed to use for example `numpy.convolve` or `scipy.ndimage.filters.convolve`. Instead, convert the formula you used in 4.1 to a vectorized version.

Name of file: `blur_2.py`, `report2.txt`

4.3 Numba implementation (6 points)

Redo 4.1, but use Numba to speed it up. Compare the runtime as before. Can you think of any advantages/disadvantages to using Numba instead of Numpy?

Name of file: `blur_3.py`, `report3.txt`

4.4 Cython implementation (IN4110 only: 10 points)

Redo your implementation in Cython, and create a report as before
Name of files: `blur_4.py`, `report4.txt`

4.5 User interface (6 points)

Provide a command line user interface for your script using `ArgumentParser` from the library `argparse`. The design of this is up to you, but it should provide instructions by calling it with a `--help` flag, and it should be possible to specify the input and output image filename. It should also be possible to switch between your 3 implementations with a command line argument.

Name of file: `blur.py`

4.6 Packaging and unit tests (6 points)

Make your implementation into a Python module or package and make a setup script. Your package should include a method `blur_image(input_filename, output_filename=None)` which returns a numpy (unsigned) integer 3D array of a blurred image of `input_filename`. If `output_filename` is supplied, the blurred image should also be saved to the specified location. You are free to make other methods which you think add useful functionality.

Further, make two unit tests. Have the tests generate a 3-dimensional numpy array with pixel values randomly chosen between 0 and 255. Assert that the maximum value of the array has decreased after blurring. The second test should choose a pixel and assert that the pixel in the blurred image is the average of its neighbors in the clear image. It is generally a good idea to fix the random seed, making debugging failing tests easier. Use the testing framework `pytest`.

Name of files: `test_blur.py`, `setup.py`

4.7 Blurring faces (5 bonus points)

Extend your script such that it can blur only a subsection of your image. You can assume this subsection has a rectangular shape.

Then create a script that detects faces and blurs only the detected faces using the package you created above. See if you can blur the faces enough (by repeated blurring) to make the face detector fail.

To detect faces, you can use OpenCV `CascadeClassifier`, using the XML file `haarcascade_frontalface_default.xml`.

```
faceCascade = cv2.CascadeClassifier("
 haarcascade_frontalface_default.xml")
faces = faceCascade.detectMultiScale(
  image,
  scaleFactor=1.025,
  minNeighbors=5,
  minSize=(30, 30)
)
```

```
print("Found {} faces!".format(len(faces)))
```

Green rectangles can be drawn around the detected faces by

```
for (x, y, w, h) in faces:
  cv2.rectangle(image, (x, y), (x + w, y + h), (0,
 255, 0), 2)
```

The values `x`, `y`, `w` and `h` can be used to define a subsection to blur. Blur these subsections and call the face detector again. Draw green rectangles on the partially blurred image where it detected faces and save it to see if you managed to trick the face detector.

Use the provided image `beatles.jpg`.

Name of file: `blur_faces.py`