1 Array Manipulation Warm-up

1.1 Exercise Skeleton

Each pattern should be implemented as a separate python class and should provide the following methods: a constructor **__init__()**, a method **draw()**, which creates the pattern using numpy functions and a visualization function **show()**. Each pattern has a public member **output**, which is an np.ndarray that stores the respective pattern.
A main script which imports and calls all these classes should also be implemented, which you can use for debugging as well. There are **no loops** needed/allowed for the creation of the patterns in this exercise! Since python is a scripting language, loops would significantly impact the performance. Also get used to proper numpy array indexing and slicing which will be tremendously important for future exercises.

Task:

- Create a file "pattern.py" and implement the classes **Checker** and **Circle** in this file. Note that we do not provide any skeleton here. Also create a file "main.py", which imports all other classes.
- Import numpy for calculation and matplotlib for visualisation using
  `import numpy as np`
  and
  `import matplotlib.pyplot as plt`.
  This is the most common way to import those packages.

Hints:
__init__() is the constructor of the class. Following functions from the cheat sheet might be useful: **np.tile()**, **np.arange()**, **np.zeros()**, **np.ones()**, **np.concatenate()** and **np.expand_dims()**

1.2 Checkerboard

The first pattern to implement is a checkerboard pattern in the class **Checkers** with adaptable tile size and resolution. You might want to start with a fixed tile size and adapt later on. For simplicity we assume that the resolution is divisible by the tile size without remainder.

Task:

- Implement the constructor. It receives two arguments: an integer **resolution** that defines the number of pixels in each dimension, and an integer **tile_size** that defines the number of pixel an individual tile has in each dimension. Store the arguments as public members. Create an additional member variable **output** that can store the pattern.
- Implement the method **draw()** which creates the checkerboard pattern as a numpy array. The tile in the top left corner should be black. In order to avoid truncated checkerboard patterns, make sure your code only allows values for **resolution** that are evenly dividable by 2· **tile_size**. Store the pattern in the public member output and return a copy. Helpful functions for that can be found on the **Deep Learning Cheatsheet** provided.
- Implement the method **show()** which shows the checkerboard pattern with for example **plt.imshow()**. If you want to display a grayscale image you can use **cmap = gray** as a parameter for this function.
- Verify your implementation visually by creating an object of this class in your main script and calling the object's functions.
- Verify your implementation by calling the unit tests with TestCheckers.

Hint: Try to build your checkerboard out of simpler constituents. Think about how tile_size and resolution must relate to each other in order to get a valid checkerboard pattern.

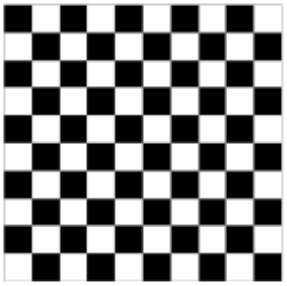

Figure 1: Checkerboard example.

1.3 Circle

The second pattern to implement is a binary circle with a given radius at a specified position in the image. Note that we expect you to use numpy operations to draw this pattern. We do not accept submissions which draw a circle with a single library function call.

Task:

- Implement the constructor. It receives three arguments: An integer **resolution**, an integer **radius** that describes the radius of the circle, and a tuple **position** that describes the position of the circle center in the image.
- Implement the method **draw()** which creates a binary image of a circle as a numpy array. Store the pattern in the public member **output** and return a copy.
- Implement the method **show()** which shows the circle with for example **plt.imshow()**.
- Verify your implementation visually by creating an object of this class in your main script and calling the object's functions.
- Verify your implementation by calling the unit tests with TestCircle.

Hints:
Think of a formula describing the circle with respect to pixel coordinates. Make yourself familiar with np.meshgrid.

Figure 2: Binary circle example.

1.4 Color Spectrum

The third pattern to implement is an RGB color spectrum. To enable the corresponding unittest, just go ahead and start implementing. Once a class **Spectrum** is defined in "pattern.py", the corresponding section in the unittests gets activated automatically.

Task:

- Implement the constructor. It receives one parameter: an integer **resolution**.
- Implement the method **draw()** which creates the spectrum in Fig. 3 as a numpy array. Remember that RGB images have 3 channels and that a spectrum consists of rising values across a specific dimension. For each color channel, the intensity minimum and maximum should be 0.0 and 1.0, respectively. Store the pattern in the public member **output** and return a copy. Hint: Particularly take a look into the corners and their color, to figure out the underlying distribution of the channels.
- Implement the method **show()** which shows the RGB spectrum with for example **plt.imshow()**.
- Verify your implementation visually by creating an object of this class in your main script and calling the object's functions.
- Verify your implementation by calling the unit tests with TestSpectrum.

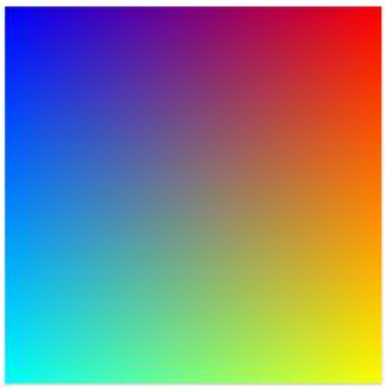

Figure 3: RGB spectrum example.