final-P2-sol

December 18, 2020

1 Academic Integrity Statement

As a matter of Departmental policy, **we are required to give you a 0** unless you **type your name** after the following statement:

> *I certify on my honor that I have neither given nor received any help, or used any non-permitted resources, while completing this evaluation.*

[TYPE YOUR NAME HERE]

2 Problem 2 (30 points)

2.1 General Note

It is possible to solve every part of this problem **without `for`-loops**. While it may be possible to appropriately use `for`-loops in this problem, solutions that use loops to achieve tasks that can be accomplished with `numpy` methods will lose points.

You might find the some of the cheatsheets to be helpful.

2.2 Prologue

The following code creates a minimal `baseArray` class that inherits from the basic `numpy` class, `np.ndarray`.

The `__new__()` method is related to `__init__()`. It describes what happens when an object is **created**, which occurs *before* that object is **initialized**. Because of the specific way in which `np.ndarray`s are constructed, it is necessary to use `__new__()` rather than `__init__()` in order to inherit from them. Because we did not discuss this in class, I have handled this for you.

Run this code now.

[3]:
```python
# run this block, do not modify
import numpy as np
class baseArray(np.ndarray):
    def __new__(cls, a):
        return np.asarray(a).view(cls)
```

2.3 Part A

Write a class called `myArray` which inherits from `baseArray`. Write code required for the following two tests to run without error:

```
# first test
x = myArray([1, 2, 3])
y = myArray([1, 1, 1])
x + y
```

```
# output
myArray([2, 3, 4])
```

```
# second test
x[1]
```

```
# output
2
```

Hint: **One** way to do this is using magic methods.

[4]:
```
# your solution here
# not necessary to do anything, since numpy arrays (and therefore baseArray)
# already support these operations
# solutions involving implementing magic methods
# only lightly penalized, provided that the implementations
# are correct.
class myArray(baseArray):
    pass
```

[5]:
```
# first test
x = myArray([1, 2, 3])
y = myArray([1, 1, 1])
x + y
```

[5]:
```
myArray([2, 3, 4])
```

[6]:
```
# second test
x[1]
```

[6]:
```
2
```

2.4 Part B

Implement the two tests above as formal unit tests and show that they run without error. It is not necessary to add a docstring to your class itself, but you should include helpful docstrings for each individual test.

[18]:
```
# your unit test class

import unittest
class testMyArray(unittest.TestCase):

    def testAddition(self):
        """
        Test for expected behavior of elementwise addition
        in the myArray class, using the example case
        provided in the problem.
        Raises an assertion error if the result of addition
        of the two sample vectors does not agree with the
        sample result.
        """
        x = myArray([1, 2, 3])
        y = myArray([1, 1, 1])
        z = myArray([2, 3, 4])

        # ok to use a for-loop here
        for i in range(2):
            self.assertEqual((x+y)[i], z[i])

        # better, if you know it
        self.assertTrue((x + y == z).all())

        # also fine
        self.assertEqual((x+y == z).mean(), 1.0)

    def testIndexing(self):
        """
        Test for expected behavior of basic indexing
        in the myArray class, using the example case provided
        in the problem.
        Raises an assertion error if the result of addition
        of the two sample vectors does not agree with the
        sample result.
        """
        x = myArray([1, 2, 3])
        self.assertEqual(x[1], 2)
```

[19]:
```
# demonstrate that your tests pass
tester = testMyArray()
tester.testAddition()
tester.testIndexing()
```

2.5 Part C

Implement a basic `Image` class that inherits from `baseArray`. Your `Image` class should accept as an initialization argument a `numpy` array with dimensions `(h, w, 3)`, where `h` and `w` are the height and width of the image in pixels, respectively. The third dimension corresponds to RGB values. All values in this array lie between 0 and 1. You may have seen image arrays like this as returned by the function `matplotlib.image.imread()` previously in the course.

Implement an `__init__()` method for your class that checks these conditions. In detail, you should check:

1. That the input is an instance of class `np.ndarray` (including its subclasses). If not, raise an informative `TypeError`.
2. That the `shape` of the input is a tuple of 3 elements. If not, raise an informative `ValueError`.
3. That the final dimension has length 3. If not, raise an informative `ValueError`.
4. That the all entries of the input are between 0 and 1, inclusive. If not, raise an informative `ValueError`.

Write simple test cases which demonstrate that your code raises the appropriate exceptions when any of these four conditions are violated. It is not necessary to write formal unit tests – just attempt to initialize your class using appropriate inputs and show the resulting error messages.

At this stage, your class doesn't need any methods other than an initializer. We'll get to the others soon.

To emphasize, your class should **not** contain an object of class `baseArray` or `np.ndarray` as an instance variable. Rather, it should **inherit** from `baseArray`.

2.5.1 Hint

We did Parts A and B for a reason.

[11]:
```
# your solution here

from matplotlib import pyplot as plt

class Image(baseArray):
    """
    A wrapper for numpy.ndarray (via the baseArray class) with support
    for several image-specific operations. The input provided to the
    class upon initialization must be an instance of np.ndarray with
    three dimensions, the third of which must have length 3.
    All entries in the supplied input must be between 0 and 1 (therefore)
    able to represent RGB values.
    """

    # Part C
    def __init__(self, x):
        """
        Check properties of the input described in the main class
        docstring, and raise informative exceptions if the required
        conditions are not met.

        x: an np.ndarray of dimension 3, with final dimension of length
            3. All entries of x must lie between 0 and 1.

        """
        if not isinstance(x, np.ndarray):
            raise TypeError("Input needs to be an instance of class np.ndarray.
↪")
        if len(x.shape) != 3:
            raise ValueError("Input must have 3 dimensions.")
        if x.shape[2] != 3:
            raise ValueError("Final dimension of input must equal 3.")
        if (x.min() < 0) or (x.max() > 1):
            raise ValueError("RGB values out of bounds; all values in input␣
↪must lie between 0 and 1")

    # Part D
    def show(self):
        """
        Display self using the imshow() method of matplotlib
        on a 7x7 axis
        """
        # adjusting the figsize is optional
        fig, ax = plt.subplots(1, figsize = (7, 7))
        ax.imshow(self)
        ax.axis("off") # this line is optional

    # Part E
    def frame(self, px = 1, color = np.array([0,0,0])):
        """
        Return a new Image consisting of self surrounded on all
        sides by a frame of the specified thickness and color.

        px: int, the thickness of the frame in pixels
        color: np.ndarray(), the color of the surrounding frame
            specified as an RGB triple.

        returns: f, a new Image consisting of the original
            image (self) surrounded by a border on all sides
            of the specified color and pixel thickness.
        """
        # this solution creates a "background" and then
        # places the original image "on top of" that background.
        # solutions that manually construct a border in some way
        # may be lightly penalized if they are significantly longer
        # or more complicated.

        s = self.shape
        f = np.zeros((s[0]+2*px, s[1]+2*px, 3))
        f[:,:] = color
        f = Image(f)
        f[px:(s[0]+px), px:(s[1]+px), :] = self
        return(f)
```

[12]:
```
# test 1
Image([1, 1, 2])
```

```
      ␣
↪---------------------------------------------------------------------------

    TypeError                                 Traceback (most recent call␣
↪last)

    <ipython-input-12-37e5bd33b3e0> in <module>
      1 # test 1
----> 2 Image([1, 1, 2])


    <ipython-input-11-2de34f08dba2> in __init__(self, x)
      8     def __init__(self, x):
      9         if not isinstance(x, np.ndarray):
---> 10             raise TypeError("Input needs to be an instance of class␣
↪np.ndarray.")
     11         if len(x.shape) != 3:
     12             raise ValueError("Input must have 3 dimensions.")


    TypeError: Input needs to be an instance of class np.ndarray.
```

[13]:
```
# test 2
Image(np.array([1, 1, 2]))
```

```
      ␣
↪---------------------------------------------------------------------------

    ValueError                                Traceback (most recent call␣
↪last)
```

```
        <ipython-input-13-f399a3a753f4> in <module>
          1 # test 2
    ----> 2 Image(np.array([1, 1, 2]))


        <ipython-input-11-2de34f08dba2> in __init__(self, x)
         10             raise TypeError("Input needs to be an instance of class␣
↪np.ndarray.")
         11         if len(x.shape) != 3:
    ---> 12             raise ValueError("Input must have 3 dimensions.")
         13         if x.shape[2] != 3:
         14             raise ValueError("Final dimension of input must equal 3.
↪")


        ValueError: Input must have 3 dimensions.
```

[14]:
```
# test 3
Image(np.array([[[1]]]))
```

```
      ␣
↪---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call␣
↪last)

        <ipython-input-14-d9d68c0984df> in <module>
    ----> 1 Image(np.array([[[1]]]))


        <ipython-input-11-2de34f08dba2> in __init__(self, x)
         12             raise ValueError("Input must have 3 dimensions.")
         13         if x.shape[2] != 3:
    ---> 14             raise ValueError("Final dimension of input must equal 3.
↪")
         15         if (x.min() < 0) or (x.max() > 1):
         16             raise ValueError("RGB values out of bounds; all values␣
↪in input must lie between 0 and 1")


        ValueError: Final dimension of input must equal 3.
```

[15]:
```
# test 4
x = np.zeros((2, 2, 3))
```

```
Image(x + 2)
```

```
␣
↪---------------------------------------------------------------------------

    ValueError                                Traceback (most recent call␣
↪last)

    <ipython-input-15-f3aa7c23bb94> in <module>
      1 x = np.zeros((2, 2, 3))
----> 2 Image(x + 2)


    <ipython-input-11-2de34f08dba2> in __init__(self, x)
     14             raise ValueError("Final dimension of input must equal 3.
↪")
     15         if (x.min() < 0) or (x.max() > 1):
---> 16             raise ValueError("RGB values out of bounds; all values␣
↪in input must lie between 0 and 1")
     17
     18     # Part D


    ValueError: RGB values out of bounds; all values in input must lie␣
↪between 0 and 1
```

3 Part D

Implement a `show()` method which visualizes the image. For example:

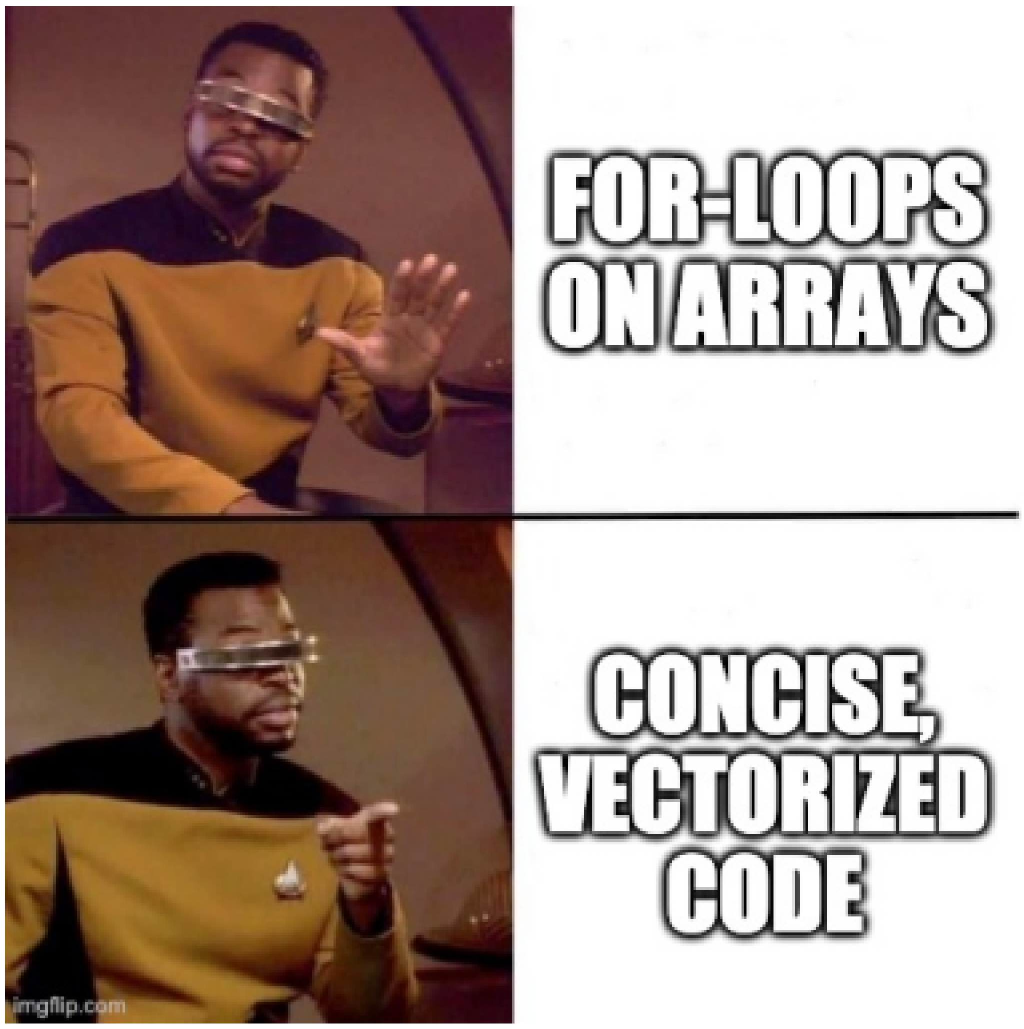
```
import matplotlib.image as mpimg
url = 'https://raw.githubusercontent.com/PhilChodrow/PIC16A/master/_images/for-loops.png'
laforge = mpimg.imread(url)

# this line is because the the image is saved in RGBA format
# not RGB. This gives it an meaningless 4th value
# for each pixel, which we discard.
laforge = laforge[:,:,:3]

im = Image(laforge)
im.show()
```

should produce

Implement the `show()` method as a clearly labeled addition to your solution from Part C. Then, run the code shown above here in Part D to demonstrate that you achieve the correct result.

- **Note**: Your output may look blurry. This is completely fine, but you may adjust the size if desired to fix this.
- **Note**: The command `ax.axis("off")` will remove frames and tickmarks from a plotting axis called `ax`. You are not required to do this, but it does look nice.

[16]:
```python
import matplotlib.image as mpimg
url = 'https://raw.githubusercontent.com/PhilChodrow/PIC16A/master/_images/
↪for-loops.png'
laforge = mpimg.imread(url)
laforge = laforge[:,:,:3]
im = Image(laforge)
im.show()
```



3.1 Part E

Give your class a method called `frame()`. The `frame()` method should return a new `Image`, in which the original `Image` is wrapped in a "frame" of user-specified thickness and color. It should accept two arguments:

1. `px`, the thickness of the frame, in pixels
2. `color`, the desired color of the frame, as an RGB triple. This should be an `np.ndarray` of shape `(3,)`.

The frame should go on the *outside* of the image, so that the entirety of the original image is still visible.

No need to copy/paste your class – just add your method to your class from Part C and then demonstrate it below. Feel free to change the color to your liking.

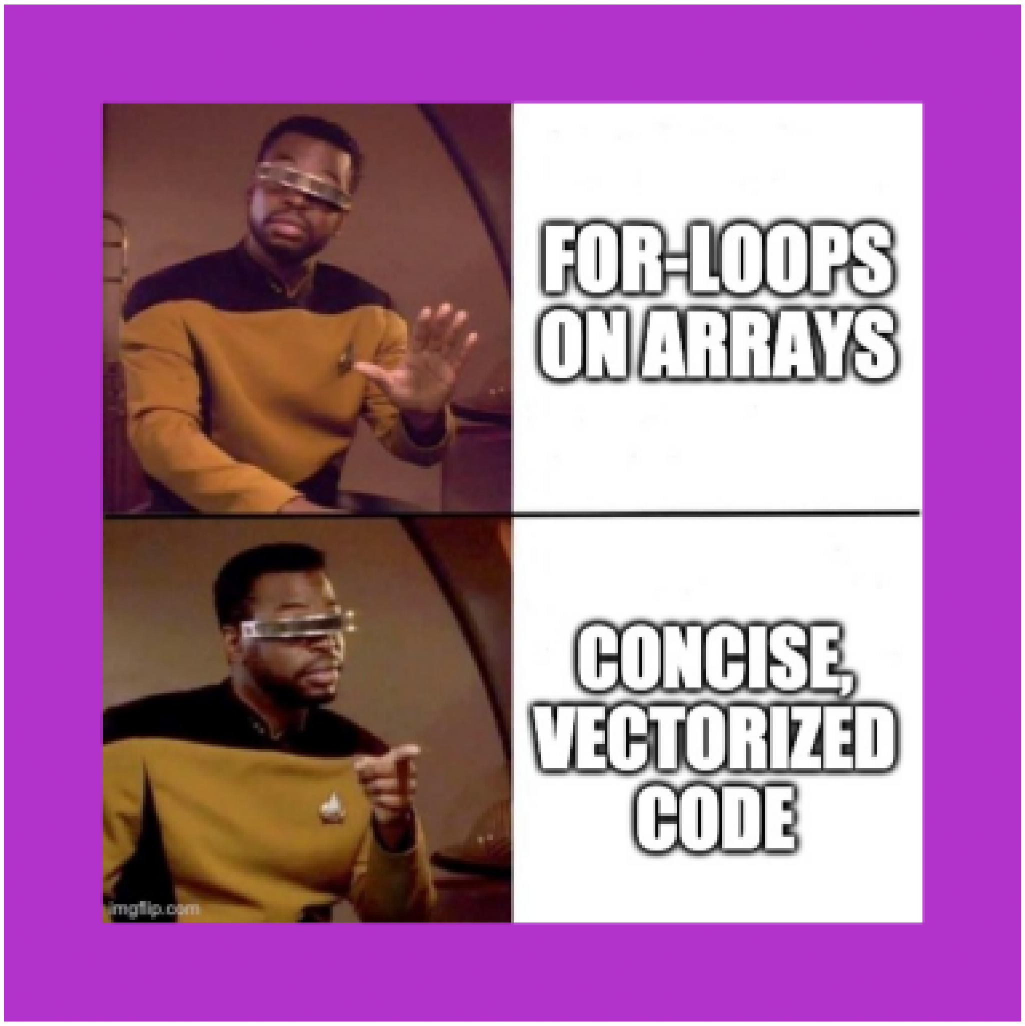
3.1.1 Example

The code

```
framed = im.frame(px = 60, color = np.array([0.7, 0.2, 0.8]))
framed.show()
```

should show the image from Part D surrounded by a thick purple frame.

```
[17]: framed = im.frame(px = 60, color = np.array([0.7, 0.2, 0.8]))
      framed.show()
```


FOR-LOOPS ON ARRAYS
CONCISE, VECTORIZED CODE
imgflip.com