Homework 4

Problem 1

Construct the following `numpy` arrays. For full credit, you should **not** use the code pattern `np.array(my_list)` in any of your answers, nor should you use `for`-loops or any other solution that involves creating or modifying the array one entry at a time.

Please make sure to show your result so that the grader can evaluate it!

```
In [16]: # run this block to import numpy
         import numpy as np
```

(A).

```
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
```

Your Solution

```
In [8]: np.reshape(np.arange(10), (5,2))
        #arrange to get 0 - 9 and then reshape for correct format
```

```
Out[8]: array([[0, 1],
               [2, 3],
               [4, 5],
               [6, 7],
               [8, 9]])
```

(B).

```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

Your Solution

```
In [17]: np.reshape(np.arange(10), (2,5))
         #arrange to get 0 - 9 and then reshape for correct format
```

```
Out[17]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])
```

(C).

```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

Your Solution

```
In [18]: np.linspace(0,1, 11)
         #there are 11 values including 0 and 1
```

```
Out[18]: array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

(D).

```
array([ 1,  1,  1,  1,  1, 10, 10, 10, 10, 10])
```

Your Solution

```
In [19]: A = np.arange(10) #get 10 values
         A[A<5]= 1 #first 5 values are 1
         A[A>4]= 10 #second 5 values are 10
         A
```

```
Out[19]: array([ 1,  1,  1,  1,  1, 10, 10, 10, 10, 10])
```

(E).

```
array([[30,  1,  2, 30,  4],
       [ 5, 30,  7,  8, 30]])
```

Your Solution

```
In [20]: B = np.arange(10) #get 10 values
         B[B%3 == 0] = 30 #values divisble by 3 turn into 30
         B = np.reshape(B, (2, 5)) #format rows and cols
         B
```

```
Out[20]: array([[30,  1,  2, 30,  4],
               [ 5, 30,  7,  8, 30]])
```

(F).

```
array([[ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

Your Solution

```
In [21]: #linspace to create values 5 - 14
         #reshape to get into right format
         C = np.reshape(np.linspace(5,14,10), (2,5))
         C.astype(int)  #use astype to change from float to int
```

```
Out[21]: array([[ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14]])
```

(G).

```
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11],
       [13, 15],
       [17, 19]])
```

Your Solution

```
In [22]: D = np.linspace(1,19,10) #get every odd num 1 - 19
         D = np.reshape(D, (5,2)) #reshape
         D.astype(int) #change from float to int
```

```
Out[22]: array([[ 1,  3],
               [ 5,  7],
               [ 9, 11],
               [13, 15],
               [17, 19]])
```

Problem 2

Consider the following array:

```
A = np.arange(12).reshape(4, 3)
A

array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

Construct the specified arrays by indexing `A`. For example, if asked for `array([0, 1, 2])`, a correct answer would be `A[0,:]`. Each of the parts below may be performed in a single line.

```
In [23]: # run this block to initialize A
         A = np.arange(12).reshape(4, 3)
         A

Out[23]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11]])
```

(A).

```
array([6, 7, 8])
```

Your Solution

```
In [24]: A[2,:] #row index 2

Out[24]: array([6, 7, 8])
```

(B).

```
array([5, 8])
```

Your Solution

```
In [25]: A[1:3, 2] #row indices 1 and 2 from col 2

Out[25]: array([5, 8])
```

(C).

```
array([ 6,  7,  8,  9, 10, 11])
```

Your Solution

```
In [26]: A[A>5] #get nums bigger than 5

Out[26]: array([ 6,  7,  8,  9, 10, 11])
```

(D).

```
array([ 0,  2,  4,  6,  8, 10])
```

Your Solution

In [27]: `#only want even numbers`
`A[A%2 == 0]`

Out[27]: `array([ 0,  2,  4,  6,  8, 10])`

(E).

`array([ 0,  1,  2,  3,  4,  5, 11])`

Your Solution

In [165]:
```
A[(A<6) | (A>10)]
#get values less than 6 and greater than 10

'''another way
np.append(A[A<6], A[A>10])
#append allows us to have them both in 1 array
'''
```

Out[165]: `array([ 0,  1,  2,  3,  4,  5, 11])`

(F).

`array([ 4, 11])`

Your Solution

In [166]:
```
A[(A == 4) | (A== 11)]
#get values 4 and 11

''' another way!
np.append(A[1,1], A[3,2])
#get indices and append values to array
'''
```

Out[166]: `array([ 4, 11])`

Problem 3

In this problem, we will use `numpy` array indexing to repair an image that has been artificially separated into multiple pieces. The following code will retrieve two images, one of which has been cutout from the other. Your job is to piece them back together again.

You've already seen `urllib.request.urlopen()` to retrieve online data. We'll play with `mpimg.imread()` a bit more in the future. The short story is that it produces a representation of an image as a `numpy` array of `RGB` values; see below. You'll see `imshow()` a lot more in the near future.

In [3]:
```python
import matplotlib.image as mpimg
from matplotlib import pyplot as plt
import urllib

f    = urllib.request.urlopen("https://philchodrow.github.io/PIC16A/homework/main.jpg")
main = mpimg.imread(f, format = "jpg").copy()

f     = urllib.request.urlopen("https://philchodrow.github.io/PIC16A/homework/cutout.jpg")
cutout= mpimg.imread(f, format = "jpg").copy()

fig, ax = plt.subplots(1, 2)
ax[0].imshow(main)
ax[1].imshow(cutout)
```

Out[3]: <matplotlib.image.AxesImage at 0x7fe40dc3cb20>
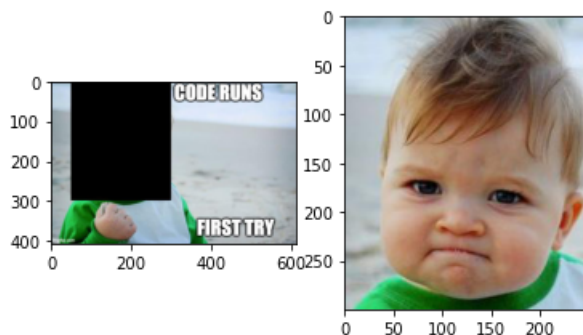
The images are stored as two `np.array`s `main` and `cutout`. Inspect each one. You'll observe that each is a 3-dimensional `np.array` of shape `(height, width, 3)`. The `3` in this case indicates that the color of each pixel is encoded as an RGB (Red-Blue-Green) value. Each pixel has one RGB value, each of which has three elements.

Use array indexing to fix the image. The result should be that array `main` also contains the data for the face. This can be done just a few lines of carefully-crafted `numpy`. Once you're done, visualize the result by running the indicated code block.

The black region in `main` starts at row 0, and column 50. You can learn more about its shape by inspecting the shape of `cutout`.

In [6]:
```python
# your solution
main[0:300, 50:300] = cutout
#place the cutout in the right dimensions
#main pic is missing 0:300 in rows and 50:300 in cols
#can leave out last arg because colors are the same
```

In [7]:
```python
# run this block to check your solution
plt.imshow(main)
```

Out[7]: <matplotlib.image.AxesImage at 0x7fe40e386130>

Problem 4

(A).

Read [these notes (https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html)](https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html) from the Python Data Science Handbook on *array broadcasting*. Broadcasting refers to the automatic expansion of one or more arrays to make a computation "make sense." Here's a simple example:

```
a = np.array([0, 1, 2])
b = np.ones((3, 3))
a.shape, b.shape

((3,), (3, 3))

a+b

array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
```

What has happened here is that the first array `a` has been "broadcast" from a 1d array into a 2d array of size 3x3 in order to match the dimensions of `b`. The broadcast version of `a` looks like this:

```
array([[0., 1., 2.],
       [0., 1., 2.],
       [0., 1., 2.]])
```

Consult the notes above for many more examples of broadcasting. Pay special attention to the discussion of the [rules of broadcasting (https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html#Rules-of-Broadcasting)](https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html#Rules-of-Broadcasting).

(B).

Review, if needed, the `unittest` module for constructing automated unit tests in Python. You may wish to refer to the [required reading (https://docs.python.org/3/library/unittest.html)](https://docs.python.org/3/library/unittest.html) from that week, the [lecture notes (https://nbviewer.jupyter.org/github/PhilChodrow/PIC16A/blob/master/content/IO_and_modules/modules/unit_testing.ipynb)](https://nbviewer.jupyter.org/github/PhilChodrow/PIC16A/blob/master/content/IO_and_modules/modules/unit_testing.ipynb), or the recorded [lecture video (https://www.youtube.com/watch?v=TwOmk9oSaR8&feature=youtu.be)](https://www.youtube.com/watch?v=TwOmk9oSaR8&feature=youtu.be).

(C).

Implement an automated test class called `TestBroadcastingRules` which tests the three rules of array broadcasting.

> **Rule 1**: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.

To test this rule, write a method `test_rule_1()` that constructs the arrays:

```
a = np.ones(3)
b = np.arange(3).reshape(1, 3)
c = a + b
```

Then, within the method, check (a) that the `shape` of `c` has the value you would expect according to Rule 1 and (b) that the final entry of `c` has the value that you would expect. **Note:** you should use `assertEqual()` twice within this method.

In a docstring to this method, explain how this works. In particular, explain which of `a` or `b` is broadcasted, and what its new shape is according to Rule 1. You should also explain the value of the final entry of `c`.

> **Rule 2**: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.

To test this rule, write a method `test_rule_2()` that constructs the following two arrays:

```
a = np.ones((1, 3))
b = np.arange(9).reshape(3, 3)
c = a + b
```

Then, within the method, check (a) that the `shape` of `c` has the value you would expect according to Rule 2 and (b) that the entry `c[1,2]` has the value that you would expect. You should again use `assertEqual()` twice within this method.

In a docstring to this method, explain how this works. In particular, explain which of `a` or `b` is broadcasted, and what its new shape is according to Rule 2. You should also explain the value of the entry `c[1,2]`.

> **Rule 3**: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

To test this rule, write a method `test_rule_3` that constructs the arrays

```
a = np.ones((2, 3))
b = np.ones((3, 3))
```

It should then attempt to construct `c = a + b`. The test should *pass* if the Rule 3 error is raised, and fail otherwise. You will need to figure out what kind of error is raised by Rule 3 (is it a `TypeError`? `ValueError`? `KeyError`?). You will also need to handle the error using the `assertRaises()` method as demonstrated in the readings.

In a docstring to this method, explain why an error is raised according to Rule 3.

You should be able to perform the unit tests like this:

```
tester = TestBroadcastingRules()
tester.test_rule_1()
tester.test_rule_2()
tester.test_rule_3()
```

Your tests have passed if no output is printed when you run this code.

Your Solution

```python
In [96]: import unittest

         # write your tester class here
         class TestBroadcastingRules(unittest.TestCase):
             '''
             Class implemented to test the three rules of array roadcasting
             '''

             def test_rule_1(self):
                 '''
                 Testing 1st rule of array broadcasting
                 a is (3,) and b is (1,3)
                 Therefore, a is padded on the left in order to match b.
                 Once a is padded we will be able to add a and b with vector addition.
                 The ending dimensions for c will be (1,3), just like b.
                 '''
                 #constructing our arrays
                 a = np.ones(3)
                 b = np.arange(3).reshape(1, 3)
                 c = a + b
                 #using assert statements to check shape of c
                 self.assertEqual(np.shape(c), (1,3))
                 #checking last entry of c
                 self.assertEqual(c[-1][-1], 3)

             def test_rule_2(self):
                 '''
                 Testing 2nd rule of array broadcasting
                 a is (1,3) and b is (3,3)
                 Therefore, a is stretched in order to match b.
                 We need to bring a from (1,3) to (3,3).
                 Once a is stretched we will be able to add a and b with vector addition.
                 The ending dimensions for c will be (3,3), just like b.

                 '''
                 #constructing arrays
                 a = np.ones((1, 3))
                 b = np.arange(9).reshape(3, 3)
                 c = a + b
                 #using assert statements to check shape of c
                 self.assertEqual(np.shape(c), (3,3))
                 #checking entry c[1][2]
                 self.assertEqual(c[1][2], 6)

             def test_rule_3(self):
                 '''
                 Testing 3rd rule of array broadcasting
                 With this case, we cannot use rule 1 to pad because both have defined
                 dimensions and no one is missing a dimension. We also cannot stretch
                 with rule 2 because both already have defined dimensions. Therefore,
                 when we add the arrays together, we will result in a ValueError because
                 there will be a conflict with the missing dimensions and neither of the
                 dimensions are 1.

                 '''
                 #constructing arrays
                 a = np.ones((2, 3))
                 b = np.ones((3, 3))
                 #using assert statment to raise value error
                 #raise value when array different sizes
                 with self.assertRaises(ValueError):
                     c = a + b
                     #adding will raise an error because we cannot add array with
                     #  different dimensions
```

```
In [97]:  # run your tests
          # your tests have passed if no output or errors are shown.

          tester = TestBroadcastingRules()
          tester.test_rule_1()
          tester.test_rule_2()
          tester.test_rule_3()
```

Problem 5

Recall the simple random walk. At each step, we flip a fair coin. If heads, we move "foward" one unit; if tails, we move "backward."

(A).

Way back in Homework 1, you wrote some code to simulate a random walk in Python.

Start with this code, or use posted solutions for HW1. If you have since written random walk code that you prefer, you can use this instead. Regardless, take your code, modify it, and enclose it in a function `rw()`. This function should accept a single argument `n`, the length of the walk. The output should be a list giving the position of the random walker, starting with the position after the first step. For example,

```
rw(5)
[1, 2, 3, 2, 3]
```

Unlike in the HW1 problem, you should not use upper or lower bounds. The walk should always run for as long as the user-specified number of steps `n`.

Use your function to print out the positions of a random walk of length `n = 10`.

Don't forget a helpful docstring!

```
In [99]:  import random
```

```
In [127]:  # solution (with demonstration) here
           def rw(n):
               '''
               function that outputs list of position of random walker after first step
                   until we reach the desire number of steps
               '''
               step = 0; #step number
               position = 0 #current position
               myList = list() #keep track of positions

               #while loop to continue until we reach desire steps
               while step < n:
                   x = random.choice([-1,1]) #step forward or back
                   position += x #position will take the step
                   myList.append(position) #list of positions
                   step += 1 #increment to show we took a step

               return myList

           #call funciton for practice
           rw(10)
```

```
Out[127]:  [-1, -2, -3, -2, -1, 0, -1, 0, 1, 0]
```

(B).

Now create a function called `rw2(n)`, where the argument `n` means the same thing that it did in Part A. Do so using `numpy` tools. Demonstrate your function as above, by creating a random walk of length 10. You can (and should) return your walk as a `numpy` array.

Requirements:

- No for-loops.
- This function is simple enough to be implemented as a one-liner of fewer than 80 characters, using lambda notation. Even if you choose not to use lambda notation, the body of your function definition should be no more than three lines long. Importing `numpy` does not count as a line.
- A docstring is required if and only if you take more than one line to define the function.

Hints:

- Check the documentation for `np.random.choice()`.
- Discussion 9, and `np.cumsum()`.

```
In [139]: # solution (with demonstration) here
          import numpy as np

          def rw2(n):
              '''
               function that outputs list of position of random walker after first step
                  until we reach the desire number of steps using numpy functions
              '''
              steps = np.random.choice([-1,1],n) #create the steps that will taken
              positionList = np.cumsum(steps) #use cumsum to show positions after steps
              return positionList

          rw2(10)
```

```
Out[139]: array([-1,  0, -1,  0,  1,  2,  3,  4,  5,  4])
```

(C).

Use the `%timeit` magic macro to compare the runtime of `rw()` and `rw2()`. Test how each function does in computing a random walk of length `n = 10000`.

```
In [144]: # solution (with demonstration) here
          %timeit rw(10000)
          %timeit rw2(10000)
```

```
6.36 ms ± 210 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
98.1 µs ± 6.1 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

(D).

Write a few sentences in which you comment on (a) the performance of each function and (b) the ease of writing and reading each function.

Part A is much slower than part B and this is also reflected in length of code. We can see that part A contains much more lines of code than part B. Also, part A uses a while loop which is a slower than using numpy functions as numpy functions are vectorized. This allows the code in part B is run on the whole vector at once instead of changing each individual index like part A. Overall we can see that part B is much faster and more efficient that part A. Both codes are fairly simple and easy to read and write. Part A is very simple and just uses a while loop to execute everything. On the other hand part B is a little more complicated as we have to understand how to use numpy functions as well as input the correct positional arguments.

(E).

In this problem, we will perform a `d`-dimensional random walk. There are many ways to define such a walk. Here's the definition we'll use for this problem:

> At each timestep, the walker takes one random step forward or backward **in each of `d` directions.**

For example, in a two-dimensional walk on a grid, in each timestep the walker would take a step either north or south, and then another step either east or west. Another way to think about is as the walker taking a single "diagonal" step either northeast, southeast, southwest, or northwest.

Write a function called `rw_d(n,d)` that implements a `d`-dimensional random walk. `n` is again the number of steps that the walker should take, and `d` is the dimension of the walk. The output should be given as a `numpy` array of shape `(n,d)`, where the `k`th row of the array specifies the position of the walker after `k` steps. For example:

```
P = rw_d(5, 3)
P

array([[-1, -1, -1],
       [ 0, -2, -2],
       [-1, -3, -3],
       [-2, -2, -2],
       [-1, -3, -1]])
```

In this example, the third row `P[2,:] = [-1, -3, -3]` gives the position of the walk after 3 steps.

Demonstrate your function by generating a 3d walk with 5 steps, as shown in the example above.

All the same requirements and hints from Part B apply in this problem as well. It should be possible to solve this problem by making only a few small modifications to your solution from Part B. If you are finding that this is not possible, you may want to either (a) read the documentation for the relevant `numpy` functions more closely or (b) reconsider your Part B approach.

```
In [182]: # solution (with demonstration) here
          def rw_d(n,d):
              '''
              will do a multidimensional step where at each timestep,
                  the walker takes one random step forward or backward
                  in each of d directions
              n = number of steps
              d = directions
              '''
              steps = np.random.choice([-1,1],(n,d)) #create steps
              positionList = np.cumsum(steps,0) #create positions
              return positionList

          rw_d(5,3)
```

```
Out[182]: array([[-1, -1,  1],
                 [ 0,  0,  2],
                 [ 1,  1,  3],
                 [ 0,  2,  2],
                 [ 1,  1,  3]])
```

(F).

In a few sentences, describe how you would have solved Part E without `numpy` tools. Take a guess as to how many lines it would have taken you to define the appropriate function. Based on your findings in Parts C and D, how would you expect its performance to compare to your `numpy`-based function from Part E? Which approach would your recommend?

Note: while I obviously prefer the `numpy` approach, it is reasonable and valid to prefer the "vanilla" way instead. Either way, you should be ready to justify your preference on the basis of writeability, readability, and performance.

In order to solve part E without the numpy tools, I think I would use another while loop encased the while loop I currently have. I would just need 2 while loops because one would go through the rows and the second would go through each column in each row. I am guessing this would take maybe 4-5 more lines as I just need another while loop and maybe define a few variables for the rows and columns. However, I know that the two while loops would take a much longer than time to process than the function that has the numpy functions. Although the numpy functions would make the code much faster, I personally would prefer to write the code with two while loops. While it would be a lot slower, I personally find it easier to visualize and understand. With the numpy functions, we had to use axis with index 0 and that personally is really difficult for me to visualize and understand. I find that without numpy functions, the code more sense intuitively.

(G).

Once you've implemented `rw_d()`, you can run the following code to generate a large random walk and visualize it.

```python
from matplotlib import pyplot as plt

W = rw_d(20000, 2)
plt.plot(W[:,0], W[:,1])
```

You may be interested in looking at several other visualizations of multidimensional random walks [on Wikipedia (https://en.wikipedia.org/wiki/Random_walk)](https://en.wikipedia.org/wiki/Random_walk). Your result in this part will not look exactly the same, but should look qualitatively fairly similar.

You only need to show one plot. If you like, you might enjoy playing around with the plot settings. While `ax.plot()` is the normal method to use here, `ax.scatter()` with partially transparent points can also produce some intriguing images.

In [162]:
```python
# solution (with demonstration) here
from matplotlib import pyplot as plt

W = rw_d(20000, 2)
plt.plot(W[:,0], W[:,1])
```

Out[162]: [<matplotlib.lines.Line2D at 0x7fe40e51d700>]