# 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. Instead use build in methods such as reshape, linspace, or arange. Use of boolean indexing may also be helpful.

### (A).

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

#### Your Solution

In [8]:
```
# run this block to import numpy
import numpy as np
```

In [38]:
```
np.arange(10).reshape(5,2)
```

Out[38]:
```
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 [37]:
```
np.arange(10).reshape(2,5)
```

Out[37]:
```
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 [10]:
```
np.arange(11)/10
```

Out[10]:
```
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 [34]:
```
np.append(np.ones((5,), dtype=int),   np.ones((5,), dtype=int) + 9)
```

Out[34]:
```
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 [40]:
```
A = np.arange(10).reshape(2,5)
A[A % 3 == 0] = 30
A
```

Out[40]:
```
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 [42]:
```
(np.arange(10)+5).reshape(2,5)
```

Out[42]:
```
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 [48]:
```
A = np.arange(20)
A[1:20:2].reshape(5,2)
```

Out[48]:
```
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.

## (A).

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

In [64]:
```
# run this block to initialize A
A = np.arange(12).reshape(4, 3)
A
```

Out[64]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

### Your Solution

In [89]:
```
A[2]
```

Out[89]:
```
array([6, 7, 8])
```

## (B).

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

### Your Solution

In [94]:
```
A[:,-1][1:3]
```

Out[94]:
```
array([5, 8])
```

## (C).

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

### Your Solution

In [95]:
```
A[A>5]
```

Out[95]:
```
array([ 6,  7,  8,  9, 10, 11])
```

### (D).

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

### Your Solution

In [96]:
```
A[A%2 ==0]
```

Out[96]:
```
array([ 0,  2,  4,  6,  8, 10])
```

### (E).

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

### Your Solution

In [117…
```
np.append(A[A<6], A[A>10])
```

Out[117…
```
array([ 0,  1,  2,  3,  4,  5, 11])
```

### (F).

```
array([ 4, 11])
```

### Your Solution

In [119…
```
np.append(A[1,1], A[3,2])
```

Out[119…
```
array([ 4, 11])
```

# Problem 3

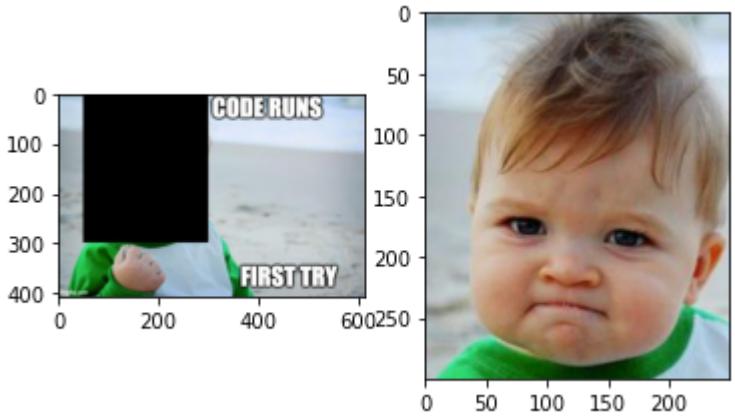
In this problem, we will use `numpy` array indexing to repair an image that has been artificially separated into multiple pieces. The following code will retrieve two images, one of which has been cutout from the other. Your job is to piece them back together again.

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. To download the images, run the code below. (You just need to press enter.)

In [2]:
```
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.jp
cutout= mpimg.imread(f, format = "jpg").copy()
```

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

Out[2]: <matplotlib.figure.AxesImage at 0x17c3afc4730>




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.

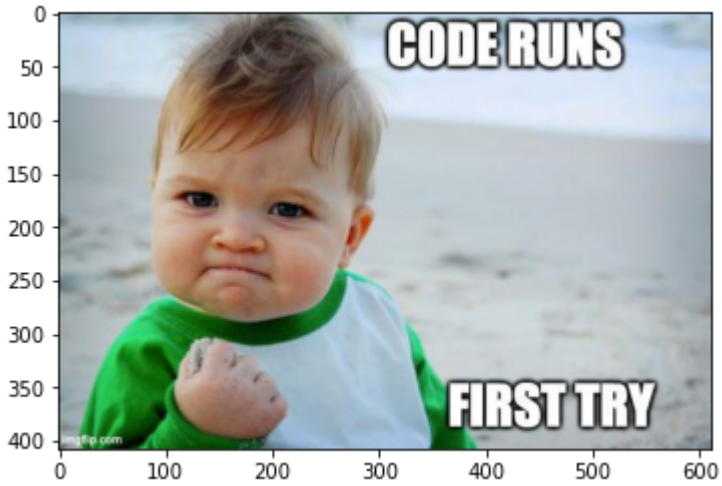
Use array indexing to fix the image. The result should be that array `main` also contains the data for the face. This can be done just a few lines of carefully-crafted `numpy`. Once you're done, visualize the result by running the indicated code block.

**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 [14]:
```
# your solution
main[0:300, 50:300] = cutout
```

In [16]:
```
# run this block to check your solution
plt.imshow(main)
```

Out[16]: <matplotlib.figure.AxesImage at 0x17c3b0d0cd0>



# Problem 4

## (A). Read about array broadcasting

One particularly nice thing about numpy arrays is that you can (sometimes) add them, even if they are different shapes. This is called *array* broadcasting. The three rules of broadcasting are:

1) If one array has fewer dimenensions than the other, is one with fewer dimensions is padded with ones **on the left**.

**Example:** If a has shape 3 and b has shape 1 by 3 and you try to type a+b, it will automatically pad a 1 to the start of a so that both arrays have shape 1 by 3.

2) Suppose two arrays have the same number of dimensions, but different shapes. If one of shapes has length one along a given dimension, then that array is "stretched out" along that direction.

**Example:** Suppose A=array([1,2]) (shape is 2) an B=array(([1,2],[3,4])). Numpy first converts A to an array with shape (1,2) (by rule 1) and then converts it to an array with shape (2,2) by stretching it out, so the result is array([[1,2],[1,2]]). It does this conversion, then performs the addition so the final answer is A+B=array([2,4],[4,6])

3) Suppose two arrays have the same number of dimensions, but different shapes along a given dimension. If neither has length one, then there is nothing you can do and an error is raised.

**Example:** If A has shape (3,4) and B has shape(4,3) then A+B will yield an error.

For more information, and many more examples read these notes from the Python Data Science Handbook.

## (B). - Review Unittest

Review, if needed, the `unittest` module for constructing automated unit tests in Python. You may wish to refer to the required reading from that week, the lecture notes or the lecture notes/videos.

## (C). - Apply Unittest to Array Broadcasting

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

In [ ]:
```
# write your tester class here
```

In [ ]:
```
# 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. Modify your code, 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`.

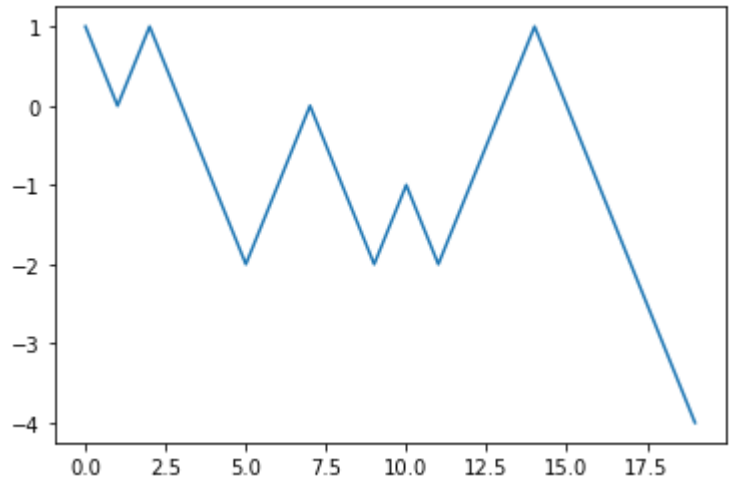
Don't forget a helpful docstring!

In [9]:
```
# solution (with demonstration) here
import random

def rw(n):

    current_position = 0
    positions = []
    coin_results = []

    while True:

        x = random.choice(["forward", "backward"])

    # if choses to move forward, add 1 to current position. log the sum on positions li
        if x == "forward":
            current_position = current_position + 1
            positions.append(current_position)
```

```
            coin_results.append("+1")

    # if choses to move backward, subtract 1 from current position. log the difference
        elif x == "backward":
            current_position = current_position - 1
            positions.append(current_position)
            coin_results.append("-1")

    # if reach upper bound or lower bound, print which reached and terminate code
        if len(positions) == n:
            print("Walk completed with length " + str(n) + ".")
            break


    print ("Positions: \n" + str(positions) + "\nCoin-Flip Moves: \n" + str(coin_result

    from matplotlib import pyplot as plt
    plt.plot(positions)
```

In [10]:
```
rw(20)
```

```
Walk completed with length 20.
Positions:
[1, 0, 1, 0, -1, -2, -1, 0, -1, -2, -1, -2, -1, 0, 1, 0, -1, -2, -3, -4]
Coin-Flip Moves:
['+1', '-1', '+1', '-1', '-1', '-1', '+1', '+1', '-1', '-1', '+1', '-1', '+1', '+1', '+
1', '-1', '-1', '-1', '-1', '-1']
```




# (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.
- Use the aggreation functions. (Not sum, but...)

In [29]:
```
# solution (with demonstration) here
def rw2(n):
    x = np.random.choice([-1,1])
    path = np.concatenate([np.zeros[1,1], x]).cumsum(0)
    start = path[:1]
    stop = path[-1:]

    print(path)
```

In [30]:
```
rw2(5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-30-62cd8aeaa4ac> in <module>
----> 1 rw2(5)

<ipython-input-29-4e829f57efe7> in rw2(n)
      2 def rw2(n):
      3     x = np.random.choice([-1,1])
----> 4     path = np.concatenate([np.zeros[1,1], x]).cumsum(0)
      5     start = path[:1]
      6     stop = path[-1:]

TypeError: 'builtin_function_or_method' object is not subscriptable
```

# (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 [32]:
```
# solution (with demonstration) here
%timeit(rw(5))
```

```
UsageError: Line magic function `%timeit(rw(5))` not found.
```

# (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.

*Your discussion here*

In [ ]:
```
```