Homework 4

Fill in your name and the names of any students who helped you below.

I affirm that I personally wrote the text, code, and comments in this homework assignment.

I received help from Matthew Chavez who gave me suggestions on Problem 2, subpart F.

- Austin Wuthrich 4/28/22

Comments are not required for problems 1 and 2. Please add comments for problem 3,4, and 5.

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 [2]:

```
# 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 [4]:

```
a = np.arange(10).reshape((5,2))
a
```

Out[4]:
```
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 [6]:
```
b=np.arange(10).reshape((2,5))
b
```

Out[6]:
```
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 [9]:
```
c=np.linspace(0,1,11)
c
```

Out[9]:
```
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 [14]: d=np.arange(11)
         mask1 = d>4
         mask2 = d<5
         d[mask1] = 10
         d[mask2] = 1
         d

Out[14]: array([ 1,  1,  1,  1,  1, 10, 10, 10, 10, 10, 10])
```

(E).

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

Your Solution

```
In [17]: e=np.arange(10).reshape((2,5))
         mask=(e%3==0)
         e[mask] = 30
         e

Out[17]: 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 [22]: f=np.arange(5,15).reshape(2,5)
         f

Out[22]: 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 [24]:
```
g=np.arange(1,20,2).reshape(5,2)
g
```

Out[24]:
```
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 [2]:
```
# run this block to initialize A
A = np.arange(12).reshape(4, 3)
A
```

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

(A).

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

Your Solution

In [27]:
```
A[2,:]
```

Out[27]:
```
array([6, 7, 8])
```

(B).

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

Your Solution

In [40]:
```
A[1:3,2]
```

Out[40]:
```
array([5, 8])
```

(C).

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

Your Solution

In [45]:
```
A[A>5]
```

Out[45]:
```
array([ 6,  7,  8,  9, 10, 11])
```

(D).

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

Your Solution

In [46]:
```
A[A%2==0]
```

Out[46]:
```
array([ 0,  2,  4,  6,  8, 10])
```

(E).

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

Your Solution

In [64]:
```
A[(A%10<6) & (A!=10)]
```

Out[64]:
```
array([ 0,  1,  2,  3,  4,  5, 11])
```

(F).

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

Your Solution

In [6]:
```
A[(A==4) | (A==11)]
```

Out[6]:
```
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.

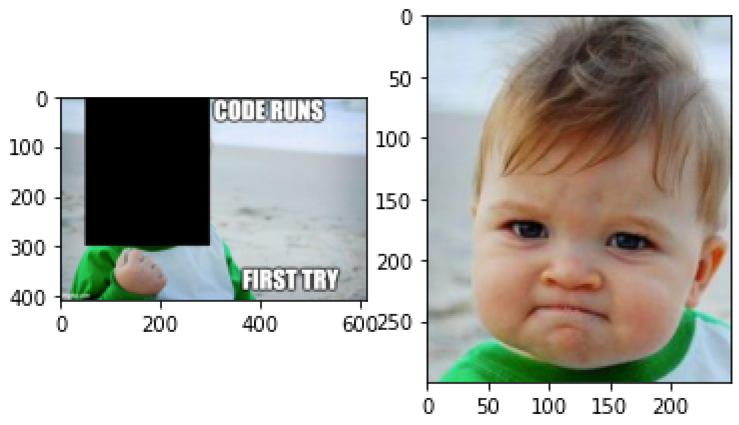
You've already seen `urllib.request.urlopen()` to retrieve online data. We'll play with `mpimg.imread()` a bit more in the future. The short story is that it produces a representation of an image as a `numpy` array of `RGB` values; see below. You'll see `imshow()` a lot more in the near future.

In [3]:
```
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 0x196327007f0>
```



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` .

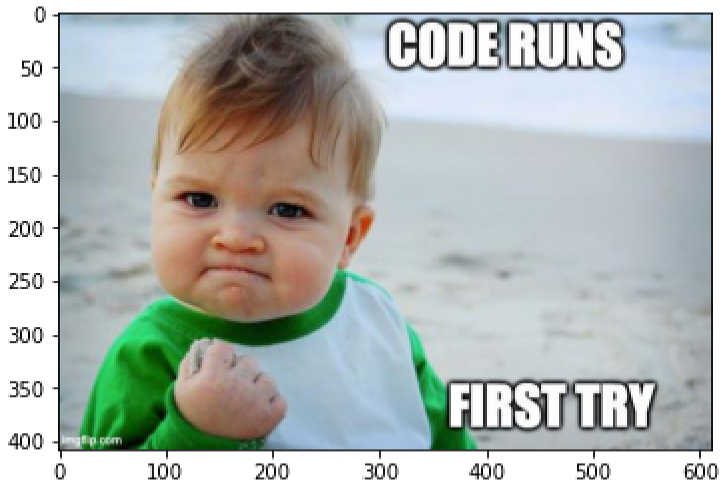
In [4]:

```
# your solution
main[0:300,50:300,:]=cutout[0:300,0:250,:]

#for indexing 3D image array, the first index represents the row of the array
#meaning the vertical height of the pixels on the image, the second index represents the columns of the array meaning the
#horizontal location of the pixels on the image, and the third index represents the color value at that image (RGB)
```

```
#For this problem, I had to inspect the main array to find the indices where the values were black and then reassign thos
#values to the cutout array values based on where the idicies started and ended

#Inspected their size and indexing below
#print(cutout[0:300,0:250,:])
#print(cutout[299:,249:,:])
```

In [5]:
```
# run this block to check your solution
plt.imshow(main)
```

Out[5]:
```
<matplotlib.image.AxesImage at 0x19632f0d490>
```



Problem 4

(A).

Read these notes 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.

(B).

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 recorded lecture video.

(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

In [17]:

```
# write your tester class here
import unittest

class TestBroadcastingRules(unittest.TestCase):
    """
    Class that tests whether numpy arrays follow the 3 broadcasting rules
    """

    def test_rule_1(self):
        """
        Function that tests whether two added numpy arrays follow broadcasting rule 1
        Param: the implicit self parameter so the function can operate using superclass TestCase

        How it works: Rule 1 works by checking for which of the arrays is 1D and then converts that one to 2D while
        while maintaining the same values and overall array structure. In this case, a is 1D and goes from [1.,1.,1.] to
        [[1.,1.,1.,]]. Notice how a after the rule 1 conversion is the same list now within another list making it 2D.
        Conveniently in this example, a and b are of the same dimensions and can now be added by corresponding entries.
        The final entry of c becomes 3.0 because a[0,2] is 1. (a float type) and b[0,2] is 2 (an int that is converted to
        float for addition). 1.+2=3.0
        """
        a = np.ones(3)
        b = np.arange(3).reshape(1, 3)
        c = a + b
        self.assertEqual(c.shape,(1,3))
        self.assertEqual(c[0,2],3.0)


    def test_rule_2(self):
        """
        Function that tests whether two added numpy arrays follow broadcasting rule 2
        Param: the implicit self parameter so the function can operate using superclass TestCase

        How it works: Rule 2 works by finding the array with one row (shape (1,x)) and then copying
        that row into as many more rows as necessary to match the number of rows of b. That way array a and b
        are of equal size (shape (3,3)) and can be added with respect to indeces. The value at c[1,2] is the
        addition of a[1,2] which is 1. and b[1,2] which is 5 that get added together into the float 6.0.
        """
        a = np.ones((1, 3))
        b = np.arange(9).reshape(3, 3)
        c = a + b
        self.assertEqual(c.shape,(3,3))
        self.assertEqual(c[1,2],6.0)


    def test_rule_3(self):
        """
        Function that tests whether two added numpy arrays follow broadcasting rule 3
        Param: the implicit self parameter so the function can operate using superclass TestCase

        How it works: Broadcasting rule 3 follows rules 1 and 2. Rule 1 cannot be applied to a or b in this case
        because neither is a 1D array. Next, python goes to rule 2 which checks if either array is 2D with only 1 row.
        In this case, a and b are 2D but do not have 1 row that can be scaled. So, python moves to rule 3. Neither array
        equals 1 and the dimension sizes differ in that b has one more row than a. As a result, python is unable to
        broadcast either array to enable addition and as such, python raises a ValueError because the arguments
        are the correct type (arrays) but the values of those arrays cannot be broadcasted and added.
        """
        a = np.ones((2, 3))
        b = np.ones((3, 3))
        with self.assertRaises(ValueError):
            c=a+b
```

In [18]:

```
# 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 [46]:

```python
# solution (with demonstration) here
def rw(n):
    """
    Function that performs a random walk for as many steps as argument specifies and creates list of position from walk
    Param n: integer representing the number of steps to take on random walk
    Returns: List containing the position of the walk after each step
    """

    import random

    #initalizing and declaring necessary variables before walk
    current_position = 0
    choice_results_log = []
    positions = []

    #walk for loop that will run for n iterations
    for i in range(n):
        x = random.choice([-1,1]) #coin-flips for int value of 1 or -1
        choice_results_log.append(x) #adds result to log of coin-flip results
        current_position+=x #updates current walk position forward one (+1) or backward one (-1)
        positions.append(current_position) #logs position change result in list

    return positions

#outputs results of walk
print("The positions of walk length 10: ", rw(10))
```

```
The positions of walk length 10:  [-1, -2, -3, -4, -3, -2, -1, 0, 1, 2]
```

(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 [41]:
```
# solution (with demonstration) here
import numpy as np

#input length of random walk: output an array representing position of walk after given step
#random.choice creates an array of length n with each position either a 1 or -1
#cumsum() then sums the cumulative value of the walk up to a given position until the end of the array/walk

rw2 = lambda n:np.cumsum(np.random.choice([-1,1],n))

rw2(10)
```

Out[41]:
```
array([-1,  0, -1, -2, -3, -2, -1,  0,  1,  2], dtype=int32)
```

(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 [47]:
```
# solution (with demonstration) here
```

```
%timeit rw(10000)
%timeit rw2(10000)
```

```
12.2 ms ± 577 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
155 µs ± 25.6 µs per loop (mean ± std. dev. of 7 runs, 1000 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.

Discussion:

A) The numpy based random walk function performed significantly quicker (155 microseconds) than the function utilizing native python lists (12.2 miliseconds). As well, the standard deviation per iteration of the function was much greater for the list-based random walk than the arrays-based random walk showing the speed and reliability of using numpy arrays.

B) The first function required significant lines of code to write even excluding the docstring as variables had to be declared and initialized, a for loop had to be constructed, and the lists/variables had to be updated each iteration in order to construct the walk. The numpy based function was easier to write after getting over the initial conceptual hill of how it could be written in a single line. Numpy's ability to construct an array from the random choice function and to then be able to cumulatively sum the array using cumsum() into another array made the walk really easy to write. In the second function it was not necessary to mess with updating variables, positions, or entries in containers because the numpy functions already had that functionality built in (and was quicker at it too).

(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 [3]:
```
# solution (with demonstration) here
import numpy as np

def rw_d(n,d):
    """
    Function that performs random walk for n paces simultaneously in d directions
    Param n: integer representing the number of steps to take on random walk
    Param d: interger representing the dimensions of the walk
    Returns: Numpy array containing the position of the walk after each step represented by each row
    """

    #create array that is the desired dimensions that can be written over
    Array_frame=np.arange(n*d).reshape(n,d)

    #create an array of desired dimensions populated with random choices
    random_choices_array = np.random.choice([-1,1],(n,d))

    #cumulatively sum the random choices to create the random walk and place entries in array frame
    return np.cumsum(random_choices_array,axis=0,out=Array_frame)

rw_d(5,3)
```

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

Discussion:

If I were to solve part E without Numpy tools, I would have had to use nested for loops. In its current random walk solution, the rw() function outputs a 1D list by using one for loop. For the part E solution, I would create a superlist that contains the sublists each representing the progress of a random walk, with each index corresponding to a certain random walk (depending on dimension). The for loops would iterate for n loops. After each loop, a new sublist containing the up to date walk positions would be added to a superlist. The for loop would update the appropriate value of each index by iterating through a list created by np.random.choice([1,-1]) and adding it to each entry of the previous sublist. Finally, the function would return the superlist.

I would recommend the numpy array option because it was much simpler than the traditional python method. As well, the numpy method allows for fewer opportunities for mistakes because there are fewer lines of code to deal with. And if a mistake is made with arrays, the errors will probably prevent the code from producing an output allowing a more precise understanding of what went wrong through the error messages. With the python option, the code may execute but have logic errors that may be more difficult to trace and fix.

(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. Your result in this part will not look exactly the same, but should look qualitatively fairly similar.

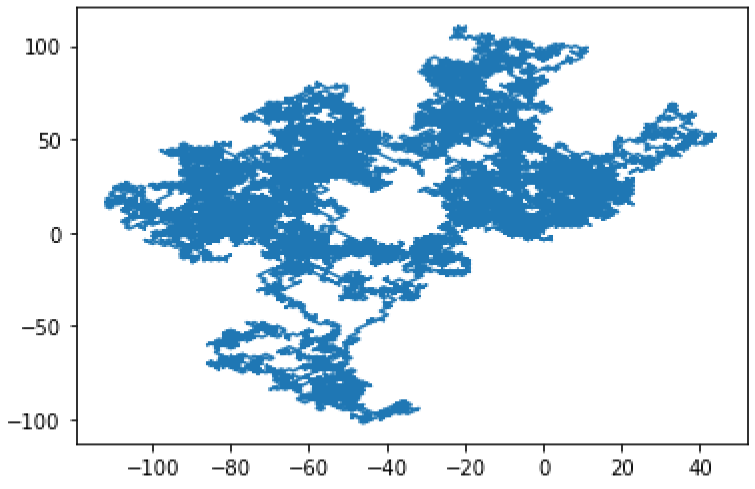
You only need to show one plot. If you like, you might enjoy playing around with the plot settings. While `ax.plot()` is the normal method to use here, `ax.scatter()` with partially transparent points can also produce some intriguing images.

In [16]:

```
# solution (with demonstration) here
from matplotlib import pyplot as plt

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

Out[16]:

```
[<matplotlib.lines.Line2D at 0x2a5844af130>]
```



In [ ]: