HW4

February 2, 2022

1 Homework 4

1.1 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!

[1]:
```
# run this block to import numpy
import numpy as np
```

1.1.1 (A).

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

Your Solution

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

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

1.1.2 (B).

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

Your Solution

[3]:
```
np.arange(10).reshape(2,5)
```

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

1.1.3 (C).

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

Your Solution

[4]:
```
np.linspace(0,1,11)
```

[4]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

1.1.4 (D).

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

Your Solution

[5]:
```
a = np.ones(10, dtype=int)
a[5:] = 10
a
```

[5]:
```
array([ 1,  1,  1,  1,  1, 10, 10, 10, 10, 10])
```

1.1.5 (E).

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

Your Solution

[6]:
```
a = np.arange(10).reshape(2,5)
a[a % 3 == 0] = 30
a
```

[6]:
```
array([[30,  1,  2, 30,  4],
       [ 5, 30,  7,  8, 30]])
```

1.1.6 (F).

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

Your Solution

[7]:
```
np.arange(5,15).reshape(2,5)
```

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

1.1.7 (G).

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

Your Solution

[8]:
```
np.linspace(1,19,10, dtype = int).reshape(5,2)
```

[8]:
```
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11],
       [13, 15],
       [17, 19]])
```

1.2 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.

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

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

1.2.1 (A).

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

Your Solution

[10]:
```
A[2,:]
```

[10]:
```
array([6, 7, 8])
```

1.2.2 (B).

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

Your Solution

[11]:
```
A[1:3,2]
```

[11]:
```
array([5, 8])
```

1.2.3 (C).

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

Your Solution

[12]:
```
np.concatenate((A[2], A[3]), axis=0)
```

[12]:
```
array([ 6,  7,  8,  9, 10, 11])
```

1.2.4 (D).

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

Your Solution

[13]:
```
A[A % 2 ==0]
```

[13]:
```
array([ 0,  2,  4,  6,  8, 10])
```

1.2.5 (E).

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

Your Solution

[14]:
```
A[(A<6) | (A==11)]
```

[14]:
```
array([ 0,  1,  2,  3,  4,  5, 11])
```

1.2.6 (F).

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

Your Solution

[15]:
```
A[(A % 7 == 4)]
```

[15]:
```
array([ 4, 11])
```

1.3 Problem 3

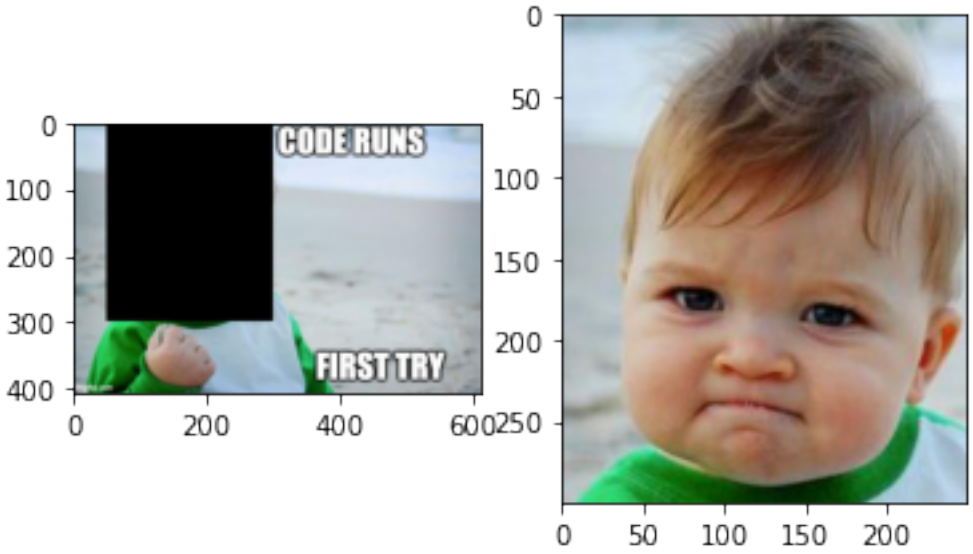
In this problem, we will use `numpy` array indexing to repair an image that has been artificially separated into multiple pieces. The following code will retrieve two images, one of which has been cutout from the other. Your job is to piece them back together again.

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.

[16]:
```
import os, ssl

if (not os.environ.get('PYTHONHTTPSVERIFY', '') and
getattr(ssl, '_create_unverified_context', None)):

    ssl._create_default_https_context = ssl._create_unverified_context
```

[17]:
```
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)
```

[17]:
```
<matplotlib.image.AxesImage at 0x23931b12370>
```



The images are stored as two `np.arrays` `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.

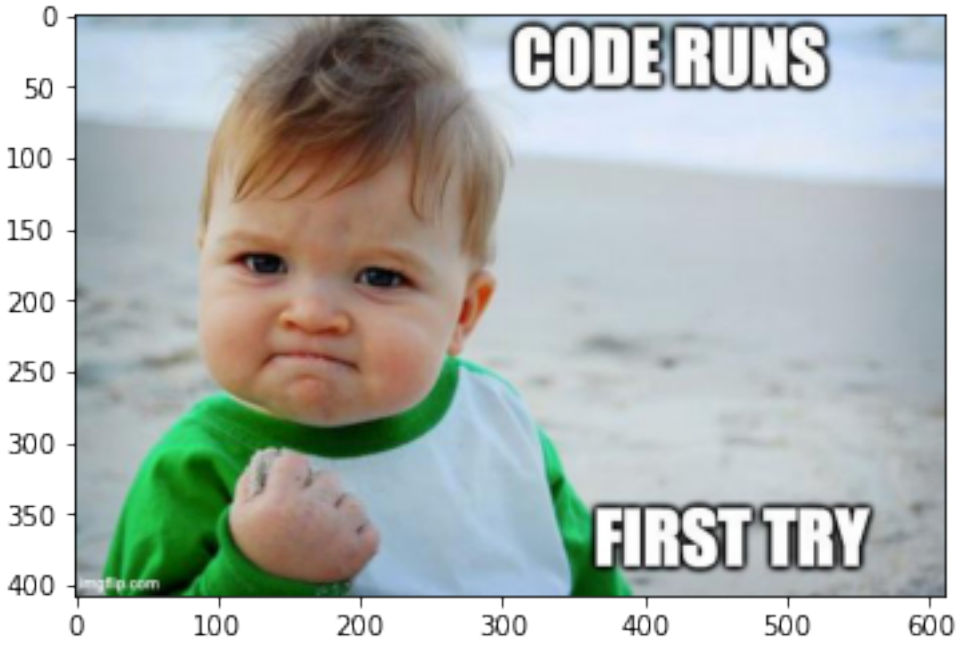
Use array indexing to fix the image. The result should be that array `main` also contains the data for the face. This can be done just a few lines of carefully-crafted `numpy`. Once you're done, visualize the result by running the indicated code block.

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`.

[18]:
```
# your solution
main[0:300, 50:300, :] = cutout[:,:,:]
```

[19]:
```
# run this block to check your solution
plt.imshow(main)
```

[19]:
```
<matplotlib.image.AxesImage at 0x239322b3520>
```



1.4 Problem 4

1.4.1 (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.

1.4.2 (B).

Review, if needed, the `unittest` module for constructing automated unit tests in Python. You may wish to refer to the https://docs.python.org/3/library/unittest.html from that week, the lecture notes, or the recorded lecture video.

1.4.3 (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.

1.4.4 Your Solution

[56]:
```
# write your tester class here
import unittest
class TestBroadcastingRules(unittest.TestCase):
    def test_rule_1(self):
        '''
        Test Rule 1 of array broadcasting.
        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.
        In this case, a should be "broadcasted" to 2d array of size 1x3 in␣
↪order to match the dimision of b.
        The broadcast version of a should be like [[1., 1., 1.]] according to␣
↪rule 1.
        The expected output of c should be like [[1.,2.,3.]] as the result of␣
↪[[1., 1., 1.]] + [[0, 1, 2]].
        '''
        a = np.ones(3) # [1.,1.,1.]
        b = np.arange(3).reshape(1, 3) # [[0,1,2]]
        c = a + b # expected to see [[1.,2.,3.]]
        self.assertEqual(c.shape, (1,3))
        self.assertEqual(c.tolist(),[[1.,2.,3.]])

    def test_rule_2(self):
        '''
        Test Rule 2 of array broadcasting.
        If the shape of the two arrays dont match in any dimension, the array␣
↪with shape equal to 1 in that dimension is stretched to match the other␣
↪shape.
        In this case, a should be "broadcasted" to 2d array of size 3x3 in␣
↪order to match the shape of b.
        The broadcast version of a should be like [[1., 1., 1.], according to␣
↪rule 2.
                                                [1., 1., 1.],
```

```
                                                    [1., 1., 1.]]
        The expected output of c should be like [[1., 2., 3.],as the result of␣
↪[[1., 1., 1.], + [[0, 1, 2],
                                                    [4., 5., 6.],              ␣
↪[1., 1., 1.],    [3, 4, 5],
                                                    [7., 8., 9.]]              ␣
↪[1., 1., 1.]]    [6, 7, 8]].

        c[1,2] is the value with the index of the second row and third column,␣
↪which should be 6. as the result of 1.0 + 5.                                  ␣
↪                                                                              ␣
↪
        '''
        a = np.ones((1, 3)) #[[1., 1., 1.]]
        b = np.arange(9).reshape(3, 3) #[[0, 1, 2],
                                       # [3, 4, 5],
                                       # [6, 7, 8]]
        c = a + b
        self.assertEqual(c.shape, (3,3))
        self.assertEqual(c[1,2], 6.)

    def test_rule_3(self):
        '''
        Test Rule 3 of array broadcasting.
        If in any dimension the sizes disagree and neither is equal to 1, an␣
↪error is raised.
        In this case, both a and b are not "broadcasted" since in 1d and 2d the␣
↪size of both disagree and neither is equal to 1.
        The error raised should be a ValueError.
        '''
        a = np.ones((2, 3)) #
        b = np.ones((3, 3)) #
        with self.assertRaises(ValueError):
            c = a + b
```

[57]:
```
# 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()
```

2 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."

2.1 (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!

[78]:
```
# solution (with demonstration) here
import random

def rw(n):
    positions = []  # history of all positions
    pos = 0         # current position
    # until n reaches 0
    while n > 0:
        # flip a coin to take step
        x = random.choice([-1, 1])
        # add result to current position p
        pos += x
        # append to position history
        positions.append(pos)
        # less step to walk
        n -=1

    return positions
```

[146]:
```
rw(10)
```

[146]:
```
[-1, -2, -1, 0, -1, 0, 1, 2, 1, 0]
```

2.2 (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()`.

[91]:
```
# solution (with demonstration) here
import numpy

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

[99]:
```
print(rw2(10))
```

```
[1 2 1 2 3 2 3 4 3 2]
```

2.3 (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`.

[100]:
```
# solution (with demonstration) here
%timeit rw(10000)
%timeit rw2(10000)
```

```
9.63 ms ± 515 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
151 µs ± 6.16 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

2.4 (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.

Using array is way more faster than using the list. The second function is precise and easy to read for people who know the python language. The first one is quiet complicate to write but is easy to read for the beginner of python.

2.5 (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.

[202]:
```
# solution (with demonstration) here
import numpy

rw_d = lambda n, d: np.cumsum(np.random.choice((-1, 1), (n,d)), axis = 0)
```

[203]:
```
p = rw_d(5,3)
p
```

[203]:
```
array([[ 1, -1,  1],
       [ 0, -2,  0],
       [ 1, -1,  1],
       [ 2, -2,  2],
       [ 3, -1,  1]], dtype=int32)
```

2.6 (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.

Without NumPy, I might write a for-loop to take the step and add them to the list. Approximately 6 lines are needed to define the function. I expect that the same situation would happen as parts a and b, NumPy tool fastens the calculation and it is preferred to use as the result of preciseness and speed.

2.7 (G).

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

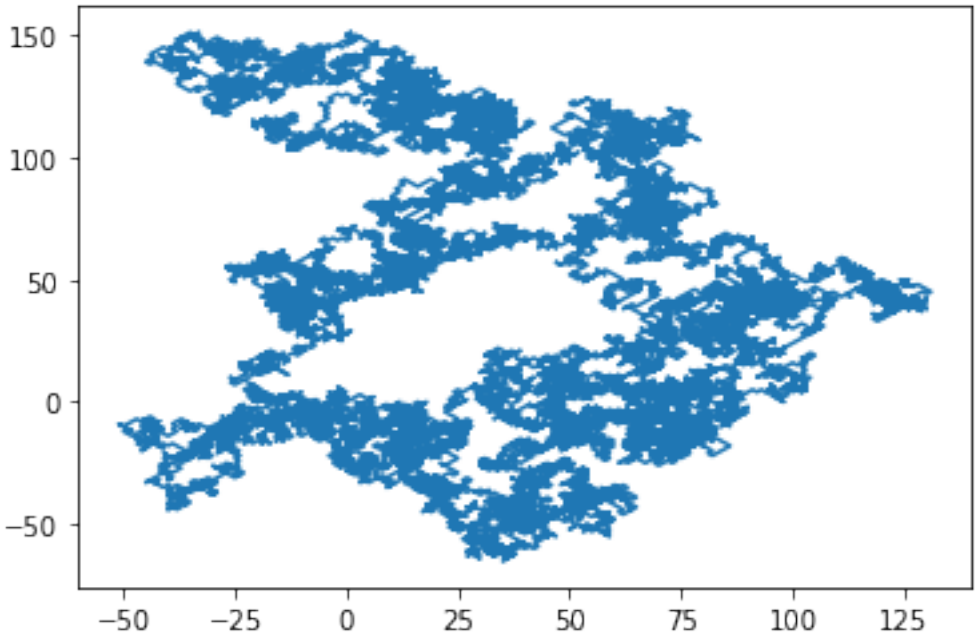
```
from matplotlib import pyplot as plt

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.

[206]:
```
# solution (with demonstration) here
W = rw_d(20000, 2)
plt.plot(W[:,0], W[:,1])
```

[206]:
```
[<matplotlib.lines.Line2D at 0x239356f8a60>]
```


150
100
50
0
−50
−50
−25
0
25
50
75
100
125