Numpy Review

In this review, we are going to refresh our memories about the Numpy package. Numpy (numerical Python) is the basic engine that turns Python into a tool for data analysis. Anything "data sciency" that you do in Python will rely on Numpy at some point, either explicitly (when calling Numpy fuctions directly), or implicitly (when, e.g., using Pandas).

The point of Numpy is do make working with data in Python easier and faster. Here, we going to remind ourselves of basic Numpy functionality.

Python lists

First, let's look at a basic Python list:

In [1]:
```
a_list = [2, 4, 6, 8]
```

Just to warm up, let's get some items from our list via *indexing*:

Get the first number from the list (the "zeroith" number in Pythonese):

In [2]:
```
a_list[0]
```

Out[2]:
```
2
```

Get the last two numbers:

In [6]:
```
print(a_list[2],a_list[3])
```

```
6 8
```

Now let's make a *nested* Python list:

In [7]:
```
a_nested_list = [[2, 4], [3, -1], [-2, 1]]
```

Remember that a Python list can hold data of different types, lengths, etc., but this list is special; it is a *list of lists* all of the same length.

Let's have a look:

In [8]:
```
print(a_nested_list)
```

```
[[2, 4], [3, -1], [-2, 1]]
```

How would we get the first entry in the second list?

We could do it in two steps... first, get the second list:

In [9]:
```
sec_list = a_nested_list[1]
```

And then get the first entry:

In [10]:
```
my_num = sec_list[0]
print(my_num)
```

```
3
```

Conveniently, we can just do this in one go:

In [11]:
```
my_num = a_nested_list[1][0]
print(my_num)
```

```
3
```

This does the same thing without having to invoke the intermediate variable `sec_list`.

While this list-of-lists construct might seem a little abstract, there is actually a nice way to wrap our heads around it, which is to think of it as a *matrix*.

> Unfortunately, no one can be told what the Matrix is. You have to see it for yourself. - *Morpheus*

Here is an example of a matrix:

 A matrix

A ***matrix*** is a 2 dimensional (2D) arrangement of things and, for our purposes, the things are data of one form or another (numbers, strings, timestamps, etc.).

These mailboxes are numbered sequentially, because there are other mailboxes in other matrices and that's just how the USPS rolls. But, notice that, for *this* matrix of mailboxes, there is another way in which we could uniquely refer to each mailbox. Specifically, we could uniquely specify each mailbox by the *row* it is in and the *column* it is in.

For example, the open mailbox with the key in the door is in the 2nd row and the 4th column or, in terms of Python indexes, the mailbox is at location [1, 3].

So, we can think of matrix as an arrangement of data that has a built in ***spatial coordinate system*** used to refer to the items of data.

Here's another Python list of lists:

In [12]:
```
another_nested_list = [[3.3, 2.3, 2.2], [1.2, 7.8, 8.7], [4.8, 2.2, 6.5],
                       [1.5, 7.5, 9.5], [5.9, 1.6, 7.7]]
```

As far as Python is concerned, this is just a list that happens to contain 5 lists, each of length 3:

In [13]:
```
print(f'Another nested Python list: {another_nested_list}')
```

```
Another nested Python list: [[3.3, 2.3, 2.2], [1.2, 7.8, 8.7], [4.8, 2.2, 6.
5], [1.5, 7.5, 9.5], [5.9, 1.6, 7.7]]
```

But it makes sense for our human brains to think about it as a 2D arrangement of data, like this:

row #	Col #					
	0	1	2	3	4	5
0	3.3	1.2	4.8	1.5	5.9	9.0
1	2.3	7.8	2.2	7.5	1.6	8.1
2	2.2	8.7	6.5	9.5	7.7	5.2

Now we can think of the Python indexes used to access the data as spatial ***row*** and ***column*** coordinates. For example:

In [14]:
```
another_nested_list[2][1]
```

Out[14]:
```
2.2
```

fetches the data value in the second row (row index 1) and the third column (column index 2).

Even though, in Python terms, `another_nested_list` is just a list of lists that all happen to be of the same length, it's very helpful for us to map data like this onto a matrix and think of the indexes as coordinates.

Numpy

Numpy is a big and powerful package, but you can think of it's most basic function as making this matrix-like way of thinking about data explicit, as opposed to just a cute way of thinking about lists of lists.

To use numpy, we first import it. Traditionally, it is imported under the name "`np`".

In [15]:
```
import numpy as np
```

Now we can convert our latest nested list into a numpy matrix using numpy's `array()` function:

In [16]:
```
our_numpy_matrix = np.array(another_nested_list)
```

Let's look at our new matrix!

In [17]:
```
print(our_numpy_matrix)
```

```
[[3.3 2.3 2.2]
 [1.2 7.8 8.7]
 [4.8 2.2 6.5]
 [1.5 7.5 9.5]
 [5.9 1.6 7.7]]
```

And compare it to the Python list from which we created it.

In [18]:
```
print(another_nested_list)
```

```
[[3.3, 2.3, 2.2], [1.2, 7.8, 8.7], [4.8, 2.2, 6.5], [1.5, 7.5, 9.5], [5.9,
1.6, 7.7]]
```

We can see that the numpy version has made the spatial row x column arrangement explicit.

Now, getting data values is easy peasy, we just index into our new matrix with row and column coordinates of a desired value:

In [19]:
```
our_numpy_matrix[2,0] # get the data at the third row and first column
```

Out[19]:
```
4.8
```

Let's see what data type our new matrix is, according to Python:

In [20]:
```
type(our_numpy_matrix)
```

Out[20]:
```
numpy.ndarray
```

So our new matrix is a Python object made by numpy of type "ndarray", which is short for "N-dimesional array" – we'll unpack this in a bit.

But our object contains other objects (in that Pythonic way), so let's see what they are:

In [21]:
```
type(our_numpy_matrix[2,0])
```

Out[21]:
```
numpy.float64
```

So they are floating point numbers, also defined in numpy, that presumably have a few more bells and whistles that regular Python floats (the 64 at the end means that 64 bits are used to store each number; this is the number's *precision*.

You may have noticed that our new matrix is a little different than the way we laid out our numbers in the table above. There, we made each sub-list into a column, whereas the `array` function seems to have made each list into a row.

Fear not! Numpy objects, like all Python objects "know" how to do things; they have *methods*. The need to turn the rows into columns and vice versa is very common – it is called *transposing* a matrix – so numpy arrays have a *transpose* method `T`.

In [22]:
```
transposed_matrix = our_numpy_matrix.T
print(transposed_matrix)
```

```
[[3.3 1.2 4.8 1.5 5.9]
 [2.3 7.8 2.2 7.5 1.6]
 [2.2 8.7 6.5 9.5 7.7]]
```

Notice that the value 4.8 used to be in the third row of the first column, but now it's coordinates have been flipped:

In [23]:
```
transposed_matrix[0,2] # new location of 4.8
```

Out[23]: 4.8

In [24]:
```
transposed_matrix[2,0] # value at the old location/
```

Out[24]: 2.2

In the code cell below, make a Python list of lists, create a numpy matrix from it, transpose it, and access 3 of it's values.

In [25]:
```
data = [[1, 2, 3],[4, 5, 6]]

matrix = np.array(data)

matrix_T = matrix.T

value1 = matrix_T[0, 0]  # 1
value2 = matrix_T[1, 0]  # 2
value3 = matrix_T[2, 1]  # 6

print(value1, value2, value3)
```

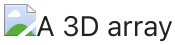
```
1 2 6
```

Numpy arrays

So far, we've been talking about our data above as a "matrix", yet we used the `array()` function to make it, and Python tells us that our matrix is an `ndarray` – what's going on?

Types of arrays

"Array" is a general term for a structured collection of data, and can have any number of dimensions (hence "ndarray" for "N-dimensional array"). Here's an (empty) 3 dimensional array:

 A 3D array

If this array were named "phred", we would index it just like above, but with an extra index – the coordinate specifying the location along the third dimension. So `phred[5, 5, 3]` would specify the bottom right location just peeking out on the fourth – what? – "page" of the array.

Though arrays can have any number of dimensions, lower-dimensional arrays are common and get their own special names.

A "matrix" is an array of 2 dimensions. As you already know, a matrix is a universal format for data and is preferably in "tidy" format, where each row is an observation and each column is a variable.

A "vector" is a list of numbers, so named because a simple list of numbers is used in math (linear algebra) and physics to specify vectors (such as force). Vectors can be

- "row vectors" - a matrix with a single row
- "column vectors" - a matrix with a single column
- a list of numbers with only a single dimension, like a Python list

Finally, in this lingo, a single number is referred to as a "scaler" (because multiplying a vector by a number scales the length of the vector without changing its direction).

In the cell below, make a Python list or tuple containing the x and y coordinates of a point (any point you like – I'm a big fan of x=3, y=1 personally). On a piece of paper or a drawing program or whatever, plot the point in an x,y coordinate system, and draw an arrow – a vector! – from the origin to your point.

In [26]:
```
v = (3, 1)
```

Now convert your Python object into a numpy array.

In [27]:
```
va = np.array(v)
```

Get the shape of your new vector using the `shape` method (used just like the `T` method above).

In [28]:
```
va.shape
```

Out[28]:
```
(2,)
```

Multiply your vector by 2 (if your vector is named "Velma", then you would literally do `Velma * 2`).

In [29]:
```
va * 2
```

Out[29]:
```
array([6, 2])
```

Plot your new vector and confirm that the multiplication *scaled* the original vector up by a factor of 2 without changing its direction!

We've just illustrated an awesome thing about numpy ndarrays: if we want to do simple operations on an entire array, we don't need to do it element-by-element, we just do it on the entire array in one go! Despite operating on arrays in general, this property is referred to as "*vectorization*" of operations.

Making numpy arrays

arrays from lists or tuples

We've already seen that we can make numpy arrays from Python lists. Like this:

In [30]:
```
print(f'A python list: {a_list}')

a_numpy_thing = np.array(a_list)

print(f'A numpy thing: {a_numpy_thing}')
```

```
A python list: [2, 4, 6, 8]
A numpy thing: [2 4 6 8]
```

Or this:

In [31]:
```
a = np.array([[1, 2, 3], [4, 5, 6]])
a
```

Out[31]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In the cell below, confirm that you can make a numpy array from a Python tuple.

In [32]:
```
data_tuple = (10, 20, 30, 40)

tupleArray = np.array(data_tuple)
```

```
print(tupleArray)
print(type(tupleArray))
```

```
[10 20 30 40]
<class 'numpy.ndarray'>
```

---

ones and zeros

If we're creating or reading in data from a source other than a Python list or tuple, we need a place to put it. To make new arrays to hold stuff, we first create an array filled with something. Most of the time, it doesn't matter what we fill it with. We commonly fill new arrays with ones or zeros.

In [33]:
```
a_vec = np.ones((3,2))
print(f'an array of ones:\n {a_vec}')
```

```
an array of ones:
 [[1. 1.]
 [1. 1.]
 [1. 1.]]
```

In the cell below, make an array of zeros.

In [35]:
```
zeroArray = np.zeros((5,5))
print(zeroArray)
```

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

Any number

We can also initialize an array to any value we want.

In [36]:
```
the_answer = np.full((3,3), 42)
print(f'Every cell has the answer!:\n {the_answer}')
```

```
Every cell has the answer!:
 [[42 42 42]
 [42 42 42]
 [42 42 42]]
```

In the cell below, get the same result as above (a 3x3 array of 42s) using 1) `np.ones` and 2) `np.zeros`. (Hint: take advantage of vectorization using `*` and `+`)

In [38]:
```
onetofortytwo = np.ones((3,3)) * 42

zerotofortytwo = np.zeros((3,3)) +42

print(onetofortytwo,zerotofortytwo)
```

```
[[42. 42. 42.]
 [42. 42. 42.]
 [42. 42. 42.]] [[42. 42. 42.]
 [42. 42. 42.]
 [42. 42. 42.]]
```

random numbers

In data science, we often add random noise to simulations in order to capture the random variability present in the universe and the data we get from it. We can do this using any number of functions in `np.random`. For example

In [39]:
```
my_noise = np.random.randn(4,4)
print(f'C\'mon feel the noise! \n {my_noise}')
```

```
C'mon feel the noise!
 [[ 0.37208846  1.43916429 -0.10593557 -0.65585162]
 [ 1.09801037  0.93998312  0.00893774  0.2933314 ]
 [-1.29555749 -1.16813431 -0.1641957  -0.09035347]
 [-0.54731316 -0.15769808  0.51952887 -0.5920664 ]]
```

Which made normally distributed (Gaussian) noise.

We can also make noise that is uniformly distributed:

In [40]:
```
unif_noise = np.random.rand(4,4)
print(f'Moar noise! \n {unif_noise}')
```

```
Moar noise!
 [[0.75587401 0.97364042 0.48392554 0.97822077]
 [0.63678057 0.80411839 0.6333357  0.42253946]
 [0.85440786 0.78264175 0.37618408 0.16576247]
 [0.65233986 0.57038772 0.55964365 0.41237328]]
```

Or we can make random integers:

In [41]:
```
int_noise = np.random.randint(1, 11, (4,4))
print(f'Random integers! \n {int_noise}')
```

```
Random integers!
 [[ 1  8  7  5]
 [ 9  2  5  8]
 [10 10  8  9]
 [ 7  7  1  9]]
```

the identity matrix

Finally, we can make an "identity matrix", a matrix with 1s running down the diagonal. It's useful for linear algebra applications, and is included here only for completeness.

In [42]:
```
aye = np.eye(4,4)
print(f'Aye aye Cap\'n! \n {aye}')
```

```
Aye aye Cap'n!
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

In the cell below, make a 6x6 array containing random integers from -10 to 10.

In [44]:
```
sixsix =  np.random.randint(-10, 10, (6,6))

print(sixsix)
```

```
[[  7   3   3 -10   7  -4]
 [  3 -10   2 -10   3   5]
 [  9  -1   2  -8  -1   8]
 [ -2  -2   3  -5   1   3]
 [ -3 -10   2  -6   6  -8]
 [ -3  -4  -8  -1   6   8]]
```

"vectorized" operations

As mentioned above, most of the common Python operators become *vectorized* in numpy, which means that we can to the same thing to every element of our arrays in one go.

let's add 11 to all our Gaussian random numbers from above.

In [45]:
```
amp_noise = my_noise + 11
amp_noise
```

Out[45]:
```
array([[11.37208846, 12.43916429, 10.89406443, 10.34414838],
       [12.09801037, 11.93998312, 11.00893774, 11.2933314 ],
       [ 9.70444251,  9.83186569, 10.8358043 , 10.90964653],
       [10.45268684, 10.84230192, 11.51952887, 10.4079336 ]])
```

Let's see which ones go to 11!

In [46]:
```
amp_noise >= 11
```

Out[46]:
```
array([[ True,  True, False, False],
       [ True,  True,  True,  True],
       [False, False, False, False],
       [False, False,  True, False]])
```

If arrays are the same size, we can do element-by-element things easily.

Which elements of the Gaussian noise are greater than the corresponding elements of the uniform noise?

In [47]:
```
my_noise > unif_noise
```

Out[47]:
```
array([[False,  True, False, False],
       [ True,  True, False, False],
       [False, False, False, False],
       [False, False, False, False]])
```

Add our Gaussian and integer noise element-by-element:

In [48]:
```
my_noise + int_noise
```

Out[48]:
```
array([[ 1.37208846,  9.43916429,  6.89406443,  4.34414838],
       [10.09801037,  2.93998312,  5.00893774,  8.2933314 ],
       [ 8.70444251,  8.83186569,  7.8358043 ,  8.90964653],
       [ 6.45268684,  6.84230192,  1.51952887,  8.4079336 ]])
```

Divide our identity matrix by our integer noise – everything off the main diagonal should be zero...

In [49]:
```
aye / int_noise
```

Out[49]:
```
array([[1.        , 0.        , 0.        , 0.        ],
       [0.        , 0.5       , 0.        , 0.        ],
       [0.        , 0.        , 0.125     , 0.        ],
       [0.        , 0.        , 0.        , 0.11111111]])
```

In the cell below, make a 10x5 matrix containing normally distributed random numbers with a mean of about 100 and a standard deviation of about 15 using vectorized operations.

In [50]:
```
matrix = np.random.normal(loc=100, scale=15, size=(10, 5))

print(matrix)
```

```
[[108.97092497  95.43016686  84.13444279  98.94630839 131.19240256]
 [ 91.65136486 114.55409771 129.78302486  97.13930683  69.89089521]
 [100.12922439  80.50349175 108.24859493  83.20234537  85.47155293]
 [102.76867263 116.3158829  104.27472572 101.96898297 110.10290861]
 [105.96793987 134.81737943  85.39739661 108.0866785  115.35693482]
 [112.39226988  98.7049404   98.28851458 106.19055134 103.55756637]
 [ 73.43635255 102.43359506 101.65511988 107.11113497  93.72227138]
 [ 99.43764531  78.57629416  98.0215138  111.51848259  68.26466912]
 [ 86.30733387  93.24802276 124.89994406  84.0866911   83.15197577]
 [122.26508324  90.26818365  91.97215481  85.17321384 110.93638278]]
```

Making numpy sequences

When computing things like functions (in the math sense), we need to start by laying down an x-axis (a working domain of the function). There are two numpy functions, `arange()` and `linspace()` that make this easy for us.

The function `arange()` allows us to specify the endpoints of our domain, and a step size (which defaults to one). We can make a sequence of the numbers one to 10 like this:

In [51]:
```
my_domain = np.arange(1,11)
print(f'my x axis is: \n {my_domain}')
```

```
my x axis is:
 [ 1  2  3  4  5  6  7  8  9 10]
```

We can also specify a step size with a third argument. Like this:

In [52]:
```
my_domain = np.arange(1,11,2)
print(f'my x axis is: \n {my_domain}')
```

```
my x axis is:
 [1 3 5 7 9]
```

The function `linspace()` is similar, but allows us to specify the number of numbers we need, and it figures out the step size for us. Like this:

In [53]:
```
my_domain = np.linspace(1,11,10)
print(f'my x axis is: \n {my_domain}')
```

```
my x axis is:
 [ 1.          2.11111111  3.22222222  4.33333333  5.44444444  6.55555556
  7.66666667  8.77777778  9.88888889 11.        ]
```

If we look at the shape of the object created by either `arange()` or `linspace()`, we see that they are 1D:

In [54]:
```
my_domain.shape
```

Out[54]:
```
(10,)
```

Note that there is only one dimension, so we only need a single index to access a value:

In [55]:
```
my_domain[3]
```

Out[55]:
```
4.333333333333334
```

We can turn this into either a row vector or a column vector by adding a second dimension. Adding a new dimension will make it, technically, a matrix – a matrix with only one column or one row, respectively.

In [56]:
```
my_domain_row = my_domain[np.newaxis,:]
print(f'my x axis row vector is: \n {my_domain_row}')
```

```
my x axis row vector is:
 [[ 1.          2.11111111  3.22222222  4.33333333  5.44444444  6.55555556
   7.66666667  8.77777778  9.88888889 11.        ]]
```

This looks the same, but let's check its shape:

In [57]:
```
my_domain_row.shape
```

Out[57]:
```
(1, 10)
```

So now it has 1 row and 10 columns. We therefore use a row *and* a column index to get a value.

In [58]:
```
my_domain_row[0, 3]
```

Out[58]:
```
4.333333333333334
```

In the cell below, make a column vector out of `my_domain`, check the shape, and get the third entry.

In [59]:
```
my_domain_col = my_domain.reshape(-1, 1)

print("Shape:", my_domain_col.shape)

third_entry = my_domain_col[2, 0]
print("Third entry:", third_entry)
```

```
Shape: (10, 1)
Third entry: 3.2222222222222223
```

Indexing cells

Cells and subsets of numpy arrays are accessed – "indexed" – much like Python lists and tuples are.

We've already done a fair amount of indexing, but let's get a bit more flexible.

Indexing rows and columns

We can fetch entire rows or columns using the the colon, `:`. Let's try this on our `my_noise` array. First, let's look at it again:

In [60]:
```
print(my_noise)
```

```
[[ 0.37208846  1.43916429 -0.10593557 -0.65585162]
 [ 1.09801037  0.93998312  0.00893774  0.2933314 ]
 [-1.29555749 -1.16813431 -0.1641957  -0.09035347]
 [-0.54731316 -0.15769808  0.51952887 -0.5920664 ]]
```

Now let's get the first column:

In [61]:
```
my_noise[:,0]
```

Out[61]:
```
array([ 0.37208846,  1.09801037, -1.29555749, -0.54731316])
```

The colon means "everything on this dimension", so the above command means "get all the rows in the first column of `my_noise`.

In the cell below, get the second (index = 1) row of data.

In [62]:
```
noiseone = my_noise[:,1]
```

Now check the shape in the cell below.

In [63]:
```
noiseone.shape
```

Out[63]:
```
(4,)
```

Now get the first row and check the shape:

In [65]:
```
noiseonerow = my_noise[0,:]
noiseonerow.shape
```

Out[65]:
```
(4,)
```

Our output in both cases above is a 1D vector. So what do we do if we want to grab a column, say, preserve it as a column? Easy! We just specify a starting and stopping index. Like this:

In [66]:
```
my_noise[:,0:1]
```

Out[66]:
```
array([[ 0.37208846],
       [ 1.09801037],
       [-1.29555749],
       [-0.54731316]])
```

You can see from the output that this is a column, but check its shape in the cell below to be sure:

In [67]:
```
my_noise[:,0:1].shape
```

Out[67]:
```
(4, 1)
```

Indexing subsets ("slicing")

What we have started doing above is called "slicing", which is carving out ("slicing") subsets of data from an array.

The key to slicing is the colon, `:`, operator.

Let's play with a 1D vector, `my_domain` first. Let's remind ourselves of it:

In [68]:
```
print(my_domain)
```

```
[ 1.          2.11111111  3.22222222  4.33333333  5.44444444  6.55555556
  7.66666667  8.77777778  9.88888889 11.        ]
```

If we put a number on either side, we can read as "from the first index to the second index":

In [69]:
```
my_domain[1:4]
```

Out[69]:
```
array([2.11111111, 3.22222222, 4.33333333])
```

If we just put an index on the left, we can read it as "from the index to the end". Like this:

In [70]:
```
my_domain[2:]
```

Out[70]:
```
array([ 3.22222222,  4.33333333,  5.44444444,  6.55555556,  7.66666667,
        8.77777778,  9.88888889, 11.        ])
```

If we just put an index on the right, we can read it as "from the beginning to the index". Like this:

In [71]:
```
my_domain[:2]
```

Out[71]:
```
array([1.        , 2.11111111])
```

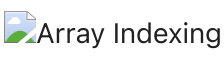
In the cell below, slice out the 3rd through 5th values of `my_domain`.

In [73]:
```
my_domain[2:5]
```

Out[73]:
```
array([3.22222222, 4.33333333, 5.44444444])
```

The extension of slicing to a matrix is straightforward. You just do your slicing on each dimension separately.

Here are some examples of array indexing from Python for Data Analysis by Wes McKinney:


Array Indexing

In the cell below, try some of these slices on `my_noise`.

In [74]:
```
my_noise[:,1:2]
```

Out[74]:
```
array([[ 1.43916429],
       [ 0.93998312],
       [-1.16813431],
       [-0.15769808]])
```

Summaries of a matrix

A numpy matrix has many methods to compute things about itself, like the sum or mean of its values.

Here's the sum of all the elements of my_noise:

In [75]:
```
my_noise.sum()
```

Out[75]:
```
-0.10606154865121642
```

Here's the arithmetic mean:

In [76]:
```
my_noise.mean()
```

Out[76]:
```
-0.006628846790701026
```

In the cell below, try `mean(0)` and `mean(1)` – what do these do?

In [78]:
```
my_noise.mean(1)
```

Out[78]:
```
array([ 0.26236639,  0.58506566, -0.67956024, -0.19438719])
```

this changes the axis in which it takes the means

In the cell below, do a dir(my_noise):

In [79]:
```
dir(my_noise)
```

Out[79]:
```
['T',
 '__abs__',
 '__add__',
 '__and__',
 '__array__',
 '__array_finalize__',
 '__array_function__',
 '__array_interface__',
 '__array_prepare__',
 '__array_priority__',
 '__array_struct__',
 '__array_ufunc__',
 '__array_wrap__',
 '__bool__',
 '__class__',
 '__class_getitem__',
 '__complex__',
 '__contains__',
 '__copy__',
 '__deepcopy__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__divmod__',
 '__dlpack__',
 '__dlpack_device__',
 '__doc__',
 '__eq__',
 '__float__',
 '__floordiv__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__iand__',
 '__ifloordiv__',
 '__ilshift__',
 '__imatmul__',
 '__imod__',
 '__imul__',
 '__index__',
 '__init__',
 '__init_subclass__',
 '__int__',
 '__invert__',
 '__ior__',
 '__ipow__',
 '__irshift__',
 '__isub__',
 '__iter__',
 '__itruediv__',
 '__ixor__',
```

```
'__le__',
'__len__',
'__lshift__',
'__lt__',
'__matmul__',
'__mod__',
'__mul__',
'__ne__',
'__neg__',
'__new__',
'__or__',
'__pos__',
'__pow__',
'__radd__',
'__rand__',
'__rdivmod__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rfloordiv__',
'__rlshift__',
'__rmatmul__',
'__rmod__',
'__rmul__',
'__ror__',
'__rpow__',
'__rrshift__',
'__rshift__',
'__rsub__',
'__rtruediv__',
'__rxor__',
'__setattr__',
'__setitem__',
'__setstate__',
'__sizeof__',
'__str__',
'__sub__',
'__subclasshook__',
'__truediv__',
'__xor__',
'all',
'any',
'argmax',
'argmin',
'argpartition',
'argsort',
'astype',
'base',
'byteswap',
'choose',
'clip',
'compress',
'conj',
'conjugate',
'copy',
'ctypes',
```

```
 'cumprod',
 'cumsum',
 'data',
 'diagonal',
 'dot',
 'dtype',
 'dump',
 'dumps',
 'fill',
 'flags',
 'flat',
 'flatten',
 'getfield',
 'imag',
 'item',
 'itemset',
 'itemsize',
 'max',
 'mean',
 'min',
 'nbytes',
 'ndim',
 'newbyteorder',
 'nonzero',
 'partition',
 'prod',
 'ptp',
 'put',
 'ravel',
 'real',
 'repeat',
 'reshape',
 'resize',
 'round',
 'searchsorted',
 'setfield',
 'setflags',
 'shape',
 'size',
 'sort',
 'squeeze',
 'std',
 'strides',
 'sum',
 'swapaxes',
 'take',
 'tobytes',
 'tofile',
 'tolist',
 'tostring',
 'trace',
 'transpose',
 'var',
 'view']
```

Now, in the cell below, see if you can use a method of `my_noise` to round all the numbers to the nearest integer:

In [80]:
```
roundednoise = np.round(my_noise)
roundednoise
```

Out[80]:
```
array([[ 0.,  1., -0., -1.],
       [ 1.,  1.,  0.,  0.],
       [-1., -1., -0., -0.],
       [-1., -0.,  1., -1.]])
```

In [ ]:
```

```