Week 1 Lab (NumPy)

COSC 3337 Dr. Rizk

Intro to NumPy

NumPy is the fundamental package for scientific computing with Python. It's used for working with arrays and contains functions for working in the domain of linear algebra, fourier transform, and matrices. In Python we have list, which serve the purpose of arrays, so why do we bother learning NumPy in the first place? Well, NumPy arrays are much faster than traditional Python lists and provide many supporting functions that make working with arrays easier. Part of why they're significantly faster is because the parts that require fast computation are written in C or C++.

Let's begin by importing NumPy and learning how to create NumPy arrays. If for some reason you don't have numpy installed, you will first have to go to your terminal (or Anaconda Prompt if on Windows) and enter the following:

```
conda install numpy
```

Make sure you've already installed Anaconda

In [1]:
```
import numpy as np
```

Creating Arrays and Common Methods

The first way of creating a NumPy array is by converting your existing Python list.

In [2]:
```
python_list = [1, 2, 3, 4, 5]
print(python_list)
print(type(python_list))
```
```
[1, 2, 3, 4, 5]
<class 'list'>
```

In [3]:
```
numpy_array = np.array(python_list)
print(numpy_array)
print(type(numpy_array))
```
```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

In [4]:
```
python_2d_array = [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]]
print(python_2d_array)
print(type(python_2d_array))
```
```
[[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]]
<class 'list'>
```

In [5]:
```
numpy_2d_array = np.array(python_2d_array)
print(numpy_2d_array)
print(type(numpy_2d_array))
```
```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]]
<class 'numpy.ndarray'>
```

However, you are more likely to use some of NumPy's built in methods to generate ndarrays. Here we'll introduce you to a few of these built in methods.

arange(start, stop, step) will return evenly spaced values within a given interval. The default step size is 1.

In [6]:
```
np.arange(0, 15)
```
Out[6]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

In [7]:
```
np.arange(0, 15, 2)
```
Out[7]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14])
```

What if we wanted a 2d array instead? We can call ***reshape(rows, columns)*** on an existing NumPy array. Please note that the product of rows and columns must evaluate to the total number of elements in your current NumPy array.

In [8]:
```
np.arange(0, 15).reshape(3, 5)
```
Out[8]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

If we'd like to generate an ndarry of zeroes or ones (useful with certain calculations), we could do so by simply calling ***zeros*** or ***ones***. For example:

In [9]:
```
np.zeros(15)
```
Out[9]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

In [10]:
```
np.zeros(15).reshape(3, 5)
```
Out[10]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

In [11]:
```
np.ones(15)
```
Out[11]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

In [12]:
```
np.ones(15).reshape(3, 5)
```
Out[12]:
```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

You might have noticed that these values defaulted to floats. If for some reason you'd like to use a different type, say, int, you can insert an additional paramater such as ***dtype=int***. See more on dtypes here

In [13]:
```
np.ones(15, dtype=int).reshape(3, 5)
```
Out[13]:
```
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])
```

A common matrix used in linear algebra is the identity matrix (an n × n square matrix with ones on the main diagonal and zeros elsewhere). We can generate this in NumPy using ***eye(n)***.

In [14]:
```
np.eye(5)
```
Out[14]:
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

Another common use case is to generate an ndarray of random numbers. This can be done in 2 ways. ***rand*** (which will fill the ndarray with random samples from a uniform distribution over [0, 1)), and ***randn*** (which will return a sample (or samples) from the standard normal distribution.) Additionally, we can use ***randint(low, high, size)*** to generate a single or multiple random integers between [low, high). The size parameter specifies how many we'd like. Let's see a few examples.

In [15]:
```
np.random.rand(5)
```
Out[15]:
```
array([0.44538557, 0.43482283, 0.20369315, 0.44795307, 0.30920162])
```

In [16]:
```
np.random.randn(5)
```
Out[16]:
```
array([-0.4242536 ,  0.71231944, -0.63292466,  0.79684565, -0.96175713])
```

In [17]:
```
np.random.randint(1,100)
```
Out[17]:
```
73
```

In [18]:
```
np.random.randint(1, 100, 15)
```
Out[18]:
```
array([99, 50, 67, 47, 50, 54, 39, 24, 55, 67, 76, 65,  8, 84, 41])
```

Other common methods that you're likely to encounter in this class include ***min***, ***max***, ***argmin***, and ***argmax***. The only difference between the two arg methods is that they'll instead return the index position of the min/max value. For example:

In [19]:
```
A = np.random.randint(0, 100, 20).reshape(4, 5)
A
```
Out[19]:
```
array([[19, 20, 74, 33, 47],
       [53, 51, 25, 78, 46],
       [ 1, 14, 41, 28, 81],
       [33, 50, 54, 67, 15]])
```

In [20]:
```
print(f'The smallest value in A is {A.min()}, and is a located at position {A.argmin(
print(f'The largest value in A is {A.max()}, and is a located at position {A.argmax()
```
```
The smallest value in A is 1, and is a located at position 10
The largest value in A is 81, and is a located at position 14
```

Lastly, we'll often find ourselves wanting to know the shape (dimensions) of our ndarray. This can be done using ***shape***.

In [21]:
```
A.shape
```
Out[21]:
```
(4, 5)
```

In [22]:
```
np.shape(A)
```
Out[22]:
```
(4, 5)
```

Great! you now know how to create NumPy arrays and some of the common methods. Let's now look into some common operations we can perform on these arrays.

Common Operations

In [23]:
```
A = np.arange(1, 16)
B = np.arange(1, 30, 2)
C = np.arange(0, 4).reshape(2, 2)
D = np.arange(0, 4).reshape(2, 2)
E = np.arange(1, 16).reshape(3, 5)
F = np.arange(11)
print(f'A: {A}')
print(f'B: {B}')
print('C:')
print(C)
print('D:')
print(D)
print('E:')
print(E)
print(f'F: {F}')
```
```
A: [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
B: [ 1  3  5  7  9 11 13 15 17 19 21 23 25 27 29]
C:
[[0 1]
 [2 3]]
D:
[[0 1]
 [2 3]]
E:
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]]
F: [ 0  1  2  3  4  5  6  7  8  9 10]
```

Arithmetic

In [24]:
```
A + B
```
Out[24]:
```
array([ 2,  5,  8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44])
```

In [25]:
```
A + 5
```
Out[25]:
```
array([ 6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
```

In [26]:
```
A - B
```
Out[26]:
```
array([  0,  -1,  -2,  -3,  -4,  -5,  -6,  -7,  -8,  -9, -10, -11, -12,
       -13, -14])
```

In [27]:
```
A - 5
```
Out[27]:
```
array([-4, -3, -2, -1,  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [28]:
```
A * B
```
Out[28]:
```
array([  1,   6,  15,  28,  45,  66,  91, 120, 153, 190, 231, 276, 325,
       378, 435])
```

In [29]:
```
A * 5
```
Out[29]:
```
array([ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75])
```

In [30]:
```
A / B
```
Out[30]:
```
array([1.        , 0.66666667, 0.6       , 0.57142857, 0.55555556,
       0.54545455, 0.53846154, 0.53333333, 0.52941176, 0.52631579,
       0.52380952, 0.52173913, 0.52      , 0.51851852, 0.51724138])
```

In [31]:
```
A / 2
```
Out[31]:
```
array([0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5,
       7. , 7.5])
```

Note: Becareful dividing by zero. You'll get ***nan*** short for Not a number.

As you may have noticed, the standard operations *, +, -, / work element-wise on arrays. If you'd like to instead do matrix multiplication, ***matmul*** can be used. Here's a quick reference in case you forgot how matrix multiplication works.

In [32]:
```
np.matmul(C, D)
```
Out[32]:
```
array([[ 2,  3],
       [ 6, 11]])
```

Universal Functions

NumPy also contains universal functions, which is a function that operates on ndarrays in an element-by-element fashion. Let's see a few examples.

In [33]:
```
np.sqrt(E)
```
Out[33]:
```
array([[1.        , 1.41421356, 1.73205081, 2.        , 2.23606798],
       [2.44948974, 2.64575131, 2.82842712, 3.        , 3.16227766],
       [3.31662479, 3.46410162, 3.60555128, 3.74165739, 3.87298335]])
```

In [34]:
```
np.log(E)
```
Out[34]:
```
array([[0.        , 0.69314718, 1.09861229, 1.38629436, 1.60943791],
       [1.79175947, 1.94591015, 2.07944154, 2.19722458, 2.30258509],
       [2.39789527, 2.48490665, 2.56494936, 2.63905733, 2.7080502 ]])
```

Something that will often come in handy is the ***where(condition, x, y)*** method. This will loop through every element in your ndarray and return a new ndarray that replaces the element with x if the condition is met, and y if the condition is not met. In the example below we're multiplying all odd numbers by 100, else replacing with -1. Try changing the -1 to F in the example below and see what happens.

In [35]:
```
np.where(F%2==0, -1, F*100)
```
Out[35]:
```
array([ -1, 100,  -1, 300,  -1, 500,  -1, 700,  -1, 900,  -1])
```

Nice! Now that you know how to create NumPy arrays and perform basic operations on them, let's take a look at how to index and select certain elements.

Indexing

Indexing 1d array

Note: recall that counting starts from 0. Given an array [5, 6, 7, 8], we say that value 5 is at index/position 0.

In [36]:
```
A = np.arange(15)
A
```
Out[36]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Obtaining a single element will look similar to Python arrays:

In [37]:
```
print(f'A[0]: {A[0]}')
print(f'A[5]: {A[5]}')
print(f'A[14]: {A[14]}')
```
```
A[0]: 0
A[5]: 5
A[14]: 14
```

We can grab a section using ***A[start_index : stop_index]***. stop_index is not inclusive.

In [38]:
```
A[0:10]
```
Out[38]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

We can also modify values in this way.

In [39]:
```
A[0:10] = 500
A
```
Out[39]:
```
array([500, 500, 500, 500, 500, 500, 500, 500, 500, 500,  10,  11,  12,
        13,  14])
```

Indexing a 2d array

In [40]:
```
B = np.arange(50).reshape(5, 10)
B
```
Out[40]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
```

Obtaining a single element can be done using ***B[row, col]***.

In [41]:
```
print(f'0th row and 2nd col in B: {B[0, 2]}')
print(f'3rd row and 3rd col in B: {B[3, 3]}')
print(f'4th row and 2nd col in B: {B[4, 2]}')
```
```
0th row and 2nd col in B: 2
3rd row and 3rd col in B: 33
4th row and 2nd col in B: 42
```

Similar to before, we can also grab a section of interest from this 2darray. Only now we have to specify both the row sections of interest and column sections of interest. ***B[0:2, 3:5]*** says: I want rows 0-2 from B, but only the elements in that row corresponding to columns 3-5. Recall that stop_index is not inclusive. Here stop_index for the rows is 2, and stop_index for the columns is 5.

In [42]:
```
B[0:2, 3:5]
```
Out[42]:
```
array([[ 3,  4],
       [13, 14]])
```

Again, we can also modify values in this way.

In [43]:
```
B[0:2, 3:5] = -1
B
```
Out[43]:
```
array([[ 0,  1,  2, -1, -1,  5,  6,  7,  8,  9],
       [10, 11, 12, -1, -1, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
```

Boolean Array Indexing

In [44]:
```
C = np.arange(50).reshape(10, 5)
C
```
Out[44]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34],
       [35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44],
       [45, 46, 47, 48, 49]])
```

A really cool feature of NumPy is that we can index arrays using comparison operators. This Lets us modify or select only elements meeting some condition. To demonstrate this, lets first see what is returned when we try to use comparison operators with our arrays.

In [45]:
```
C%2 == 0
```
Out[45]:
```
array([[ True, False,  True, False,  True],
       [False,  True, False,  True, False],
       [ True, False,  True, False,  True],
       [False,  True, False,  True, False],
       [ True, False,  True, False,  True],
       [False,  True, False,  True, False],
       [ True, False,  True, False,  True],
       [False,  True, False,  True, False],
       [ True, False,  True, False,  True],
       [False,  True, False,  True, False]])
```

We get back an array of the same shape telling us which values in ***C*** satisfy the condition C%2==0 (even values). Recall that 0 is an alias for False, and 1 is an alias for True. Because of this, we can actually call the sum function right off of this array, which will evaluate to the total number of even numbers in ***C***.

In [46]:
```
(C%2 == 0).sum()
```
Out[46]:
```
25
```

Something we'll find ourselves doing more often is passing the boolean array as an index. What this will do is filter out the False elements and only leave us with the elements corresponding to True (even values in our case).

In [47]:
```
C[C%2 == 0]
```
Out[47]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
       34, 36, 38, 40, 42, 44, 46, 48])
```

Congratulations! You now know how to create NumPy arrays, perform common operations on them, and how to index them. There's so much that NumPy can do, but this should cover just about all that you'll need to succeed in this course. Other popular Python libraries used for data science such as Pandas and Matplotlib are built on top of NumPy, so you'll be using a lot of these features alongside those libraries.