Week 1 Lab (NumPy)

COSC 3337 Dr. Rizk

This lab is worth **100 points** and is due on **Tuesday 2/11 at *11:59 pm***

The questions are **bolded** with the points being ***bolded and italicized***

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 pandas and learning about the Series data type. If for some reason you don't have pandas 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

```
import numpy as np
```

Creating Arrays and Common Methods

The first way of creating a NumPy array is by converting your existing Python list. First we need to create a python list. A python list is similar to a C++ array (except for the main difference that the python list can be modified).

A python list can also contain elements of different data types. For now, let us create a simple python list with elements of the same data type. We'll see why we do that. Recall that the syntax is as follows:

variable_name=[element1, element2...]

For this task, in the next code cell, how would you:

a)create a python list with elements of the same data type from including 1 to including 5 (as below)?

b)print the python list?

c)print the type of python list using the syntax print(type(variable_name))?

(*6 points*)

```
Start coding or generate with AI.
```

```
[1, 2, 3, 4, 5]
<class 'list'>
```

We see that the type is a 'list'. Now we would like to convert this list to a numpy array since NumPy arrays are much faster and more easier to work with. To convert a python list to a numpy array, the method is similar to type casting but much more easier. To convert a python list to a numpy array, the syntax is as follows:

new_variable=numpy.array(old_python_list_variable). Since you defined numpy as np, you can replace the word 'numpy' with 'np' to give the same result

For this task, in the next code cell:

a)convert your previous python list to a numpy array.

b)print the numpy array

c)print the type of numpy array using the syntax print(type(variable_name))

(*6 points*)

```
Start coding or generate with AI.
```

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

We now see why we created a python list with elements of the same data type. That is because numpy arrays can only contain elements of the same data type (similar to a c++ array).

Now we will do something similar for a python 2d array (lets call it an array to be consistent). The syntax is as follows: variable_name= [[element1a, element1b...], [element2a,element2b...],...]

```
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'>
```

For this task, in the next code cell:

a)create a 2d python array with elements of the same data type. Do not use the previous numbers

b)print the python array

c)print the type of python array using the syntax print(type(variable_name))

(*6 points*)

```
Start coding or generate with AI.
```

The method to convert a n-dimensional python array to a numpy array is similar to converting a 1d python array.

For this task, in the next code cell:

a)convert the *python_2d_array* variable into a numpy array.

b)print the numpy array

c)print the type of numpy array using the syntax print(type(variable_name))

(*6 points*)

```
Start coding or generate with AI.
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]]
<class 'numpy.ndarray'>
```

Creating a python array and then converting it to a numpy array can be tedious. Therefore, 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.

The first one is using ***arange(start, stop, step (optional))***.

As you can see from the syntax, it will return evenly spaced values within a given interval. The default step size is 1. A key thing to remember is that the size of the array will be stop - step (when the step size is 1). **For example, create numpy array where the start is 0 and stop is 15. Try it in the next code cell.**

(*2 points*)

```
Start coding or generate with AI.
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

As discussed above, when the step is not defined, a default of 1 is used. However, you can also define a step size.

For this task, in the next code cell:

a)create a numpy array from 0 (start) to 15 (stop) with step size of 2.

(*2 points*)

```
Start coding or generate with AI.
```

```
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.

The example below creates a 2d numpy array with step size 1 into a 3x5 matrix

```
np.arange(0, 15).reshape(3, 5)
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

In the next code cell, how would you create a 2d numpy array with step size 1 into a *5x3* matrix?

(*2 points*)

```
Start coding or generate with AI.
```

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

In the next two cells:

1)Create a numpy array of all (15) zeros

2)Create a 3x5 numpy array of a total of 15 zeros.

3,4) Do the same for 1s.

(*16 points*)

```
Start coding or generate with AI.
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
Start coding or generate with AI.
```

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

```
Start coding or generate with AI.
```

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

```
Start coding or generate with AI.
```

```
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

Below is an example of how you can create a 3x5 ones numpy array with data type integer.

```
np.ones(15, dtype=int).reshape(3, 5)
```

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

In the next code cell, how would you create a 3x3 zeros numpy array with data type integer?

(*2 points*)

```
Start coding or generate with AI.
```

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)***.

Using the previously learned syntax, how would you create a 5x5 matrix with ones on the diagonal and zeros elsewhere? (hint: this is 'n')

(*2 points*)

```
Start coding or generate with AI.
```

```
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.

```
np.random.rand(5)
```

```
array([0.44538557, 0.43482283, 0.20369315, 0.44795307, 0.30920162])
```

```
np.random.randn(5)
```

```
array([-0.4242536 ,  0.71231944, -0.63292466,  0.79684565, -0.96175713])
```

```
np.random.randint(1,100)
```

```
73
```

```
np.random.randint(1, 100, 15)
```

```
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. Given the array A for example:

```
A = np.random.randint(0, 100, 20).reshape(4, 5)
A
```

```
array([[19, 20, 74, 33, 47],
       [53, 51, 25, 78, 46],
       [ 1, 14, 41, 28, 81],
       [33, 50, 54, 67, 15]])
```

What would you replace the '???' with to get the statement below?

(*4 points*)

```
print(f'The smallest value in A is {???}, and is a located at position {???}')
print(f'The largest value in A is {???}, and is a located at position {???}')
```

```
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***.

```
A.shape
```

```
(4, 5)
```

```
np.shape(A)
```

```
(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

```
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]
```

Given the above code cell, in the next code cells, how would you:

i)Add A and B?

ii)Add 5 to A (this means for all values of A)

iii)Subtract B from A

iv)Subtract 5 from A

v)Multiply A with B

vi)Multiply A with 5

vii)Divide B from A

viii)Divide 2 from A

(*24 points*)

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

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.

```
np.matmul(C, D)
```

```
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.

```
np.sqrt(E)
```

```
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]])
```

```
np.log(E)
```

```
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? Explain why this occurs

(*2 points*)

```
np.where(F%2==0, -1, F*100)
```

```
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.

```
A = np.arange(15)
A
```

```
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.

Using the previous example, how would you complete the rest of the print statements to give the desired results as shown below?

(*2 points*)

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

```
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.

```
A[0:10]
```

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

We can also modify values in this way.

```
A[0:10] = 500
A
```

```
array([500, 500, 500, 500, 500, 500, 500, 500, 500, 500,  10,  11,  12,
            13,  14])
```

In the next cell, modify A such that the first 5 elements are 0.

(*2 points*)

```
Start coding or generate with AI.
```

Indexing a 2d array

```
B = np.arange(50).reshape(5, 10)
B
```

```
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]*. How would you complete the following print statements to show the correct answer?

(*12 points*)

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

```
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.

```
B[0:2, 3:5]
```

```
array([[ 3,  4],
       [13, 14]])
```

How would you get a subset like this?

[[5, 6],

[15, 16]]

(*2 points*)

```
Start coding or generate with AI.
```

Again, we can also modify values in this way.

```
B[0:2, 3:5] = -1
B
```

```
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

```
C = np.arange(50).reshape(10, 5)
C
```

```
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.

```
C%2 == 0
```

```
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]])
```

Double-click (or enter) to edit

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***.

```
(C%2 == 0).sum()
```

```
25
```

In the next cells, how would you:

a)Return the array where the numbers divisible by 3 are True and the rest are False? Hint see how we did C%2==0

b)Return how many True were found? Hint see how we did in the above cell.

(*2 points*)

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

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).

```
C[C%2 == 0]
```

```
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.