HW1 Notebook - Part 1

Welcome to part 1 of your second HW notebook!

Notebook Setup

In [1]:
```
mpy as np
tplotlib.pyplot as plt
aborn as sns
 additions
hon.display import HTML
 "<style>div.warn{background-color: #fcf2f2;border-color: #dFb5b4; border-le
<style>div.info{padding: 15px; border: 1px solid transparent; border-left: 5
e)
```

Out[1]:

Problem 0

As in HW0, Problem 0 on this HW requires reading and interacting with the **excellent** Python Like You Mean it (https://www.pythonlikeyoumeanit.com/) resource, specifically Module 3, introducing you to the basics of numpy.

 **Note:** The original resource is not itself interactive. For the sake of this class, I have converted the code *samples* in Module 3, to interactive code *cells* in a notebook and provide you with them for our class.

 **Warning:** This conversion was done mostly automatically by a script I wrote. I spot-checked the conversion and it looked ok. If there is anything "weird" in your notebook, please refer back to the original source online!

Please let me know by reporting it on Piazza and I will update it!

 **Note:** Not all of Module 3 is **required**. From the list below, I have denoted the **required sections** with a  emoji

Module 3: The Essentials of NumPy

-  **Introducing the ND-array** (../PLYMI/Module3/IntroducingTheNDarray.ipynb)

- ⚠ **Accessing Data Along Multiple Dimensions in an Array** (../PLYMI/Module3/AccessingDataAlongMultipleDimensions.ipynb)
  - One-dimensional Arrays
  - Two-dimensional Arrays
  - N-dimensional Arrays
  - Zero-dimensional Arrays
  - Manipulating Arrays

- ⚠ **Basic Array Attributes** (../PLYMI/Module3/BasicArrayAttributes.ipynb)

- ⚠ **Functions for Creating NumPy Arrays** (../PLYMI/Module3/FunctionsForCreatingNumpyArrays.ipynb)
  - Creating Arrays from Python Sequences
  - Creating Constant Arrays: zeros and ones
  - Creating Sequential Arrays: arange and linspace
  - Creating Arrays Using Random Sampling
  - Creating an Array with a Specified Data Type
  - Joining Arrays Together

- Iterating Over Arrays & Array-Traversal Order (../PLYMI/Module3/ArrayTraversal.ipynb)
  - How to Traverse an Array: Row-major (C) vs Column-major (F) Traversal Ordering

- ⚠ **"Vectorized" Operations: Optimized Computations on NumPy Arrays** (../PLYMI/Module3/VectorizedOperations.ipynb)
  - Basic Mathematical Operations Using Arrays
  - Vectorized Operations
  - NumPy's Mathematical Functions
  - Logical Operations
  - Linear Algebra

- ⚠ **Array Broadcasting** (../PLYMI/Module3/Broadcasting.ipynb)
  - Rules of Broadcasting
  - A Simple Application of Array Broadcasting
  - Size-1 Axes & The newaxis Object
  - An Advanced Application of Broadcasting: Pairwise Distances

- ⚠ **Introducing Basic Indexing** (../PLYMI/Module3/BasicIndexing.ipynb)
  - Basic Indexing
  - Producing a View of an Array
  - Augmenting the Underlying Data of an Array

- Advanced Indexing (../PLYMI/Module3/AdvancedIndexing.ipynb)
  - Integer Array Indexing
  - Boolean-Array Indexing
  - In-Place & Augmented Assignments via Advanced Indexing
  - Combining Basic and Advanced Indexing Schemes

Problem 1

Now that you've read and interacted with most of Module 3, you are ready to "test your skills" on this problem!

1. Generate an array of shape `3,4,10` filled with random numbers.

Hint: You can use the `np.random.random` function!

In [78]:
```
np.random.random((3,4,10))
```

Out[78]:
```
array([[[0.95865361, 0.1534306 , 0.12074118, 0.01180842, 0.99022442,
         0.77476423, 0.51777706, 0.60706607, 0.69917813, 0.04620392],
        [0.07526299, 0.48345842, 0.14021622, 0.74283972, 0.63600443,
         0.94026841, 0.75121572, 0.10819308, 0.3453073 , 0.30810719],
        [0.52705699, 0.65568165, 0.9428944 , 0.52598544, 0.95131437,
         0.24921298, 0.36136939, 0.46081276, 0.26384299, 0.30223246],
        [0.88679078, 0.0491622 , 0.99194097, 0.5663995 , 0.18656371,
         0.82886637, 0.61928727, 0.6773621 , 0.26028381, 0.26933693]],

       [[0.67098944, 0.56099237, 0.1546649 , 0.63709872, 0.17069796,
         0.50448357, 0.9818067 , 0.20628552, 0.05915272, 0.60190062],
        [0.82198538, 0.1025048 , 0.67160374, 0.96637008, 0.58970685,
         0.87136961, 0.6569493 , 0.06233619, 0.42984076, 0.27456483],
        [0.21446854, 0.6384749 , 0.85224408, 0.23470146, 0.3124883 ,
         0.98912728, 0.07701484, 0.63800363, 0.35516401, 0.30921584],
        [0.94561073, 0.48749858, 0.75976744, 0.26996005, 0.3309188 ,
         0.95333931, 0.05245064, 0.36122075, 0.30321106, 0.9467085 ]],

       [[0.84103767, 0.11455503, 0.33043483, 0.92494087, 0.91444069,
         0.71237113, 0.63306285, 0.03568318, 0.34337435, 0.33792139],
        [0.67313827, 0.43714123, 0.0939391 , 0.29257074, 0.09263038,
         0.77576466, 0.5953683 , 0.91838866, 0.17843584, 0.58777541],
        [0.50691798, 0.6239404 , 0.99560792, 0.20579676, 0.25398923,
         0.86964238, 0.29193978, 0.74166692, 0.21422704, 0.72947042],
        [0.87523654, 0.20696612, 0.07563491, 0.73569633, 0.92008872,
         0.2581454 , 0.00377021, 0.81613079, 0.17903324, 0.15173129]]])
```

Type *Markdown* and LaTeX: α^2

2. Generate an array of 47 random integers from the range 1-100.

In [80]:
```
np.random.randint(1,100,size=(47,))
```

Out[80]:
```
array([80,  1, 86, 31, 90, 31,  7, 32, 27, 64, 34, 45, 37, 94, 46, 73, 9
3,
       90, 82, 17, 63, 75, 32, 50, 11, 51, 64,  5,  6, 87, 98, 13, 35,
1,
       98, 41, 41, 43, 78, 58, 24, 55, 56, 82, 24, 70, 12])
```

3. Create the following arrays:

```
x_1 = [1,2,3,4,5]
y_2 = [99, 100, 20, 23, 45]
```

then combine them into a matrix `A` (in numpy, just another ND-array) of shape `(2,5)`.

In [85]:
```
x_1 = np.array([1,2,3,4,5])
y_1 = np.array([99, 100, 20, 23, 45])
```

In [86]:
```
A = np.vstack([x_1,y_1])
A
```

Out[86]:
```
array([[  1,   2,   3,   4,   5],
       [ 99, 100,  20,  23,  45]])
```

4. Add a column of `1` s to the "front" of your new matrix `A`. That is, make `A` have shape `(2,6)` and A[:,0] be all equal to 1.

In [87]:
```
ones = np.ones((A.shape[0],1), dtype=int)
```

In [88]:
```
A = np.hstack([ones,A])
A
```

Out[88]:
```
array([[  1,   1,   2,   3,   4,   5],
       [  1,  99, 100,  20,  23,  45]])
```

5. Slice your matrix `A` to grab all rows and columns 1-3.:

In [72]:
```
B = A[:,0:3]
B
```

Out[72]:
```
array([[  1,   1,   2],
       [  1,  99, 100]])
```

6. Using numpys `arange` and `reshape` functions, generate an array of integers from -5 to 20 (not including 20), and turn them into a `(5,5)` matrix, called `F`

In [98]:
```
F = np.arange(-5,20).reshape(5,5)
F
```

Out[98]:
```
array([[-5, -4, -3, -2, -1],
       [ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

7. Given this matrix `F` you just created, take the element-wise sum of the 1st and 3rd rows together (producing a single value)

In [99]:
```
np.sum(F[0,:] + F[2,:])
```

Out[99]: 20

8. Write a new function called `myMSE` which takes two vectors `a` and `b` and calculates:

$$\|a-b\|_2^2 = \sum_{i=0}^{d}(a_i - b_i)^2$$

without using any for loops.

Test in on the following inputs:

```
vec1 = [1,2,3]
vec2 = [4,5,6]
print(myMSE(vec1,vec2))
---> 27
```

In [104]:
```
def myMSE(a,b):
    a = np.array(a)
    b = np.array(b)
    return np.sum(np.square(a-b))
```

In [105]:
```
vec1 = [1,2,3]
vec2 = [4,5,6]
print(myMSE(vec1,vec2))
```

```
27
```

9. For this problem, lets look at the dot product!

- Create an array `a_1` with the integers between `1-10` (including 10).
- Create an array `b_1` with the integers between `11-20` (including 20).
- Take their **dot product** using numpy's `np.dot`.
- Compare this to you calculating it by hand.
- Now re-calculate taking advantage of the `.dot` function built into arrays themselves! **Hint**. Try typing `a_1.dot?`.

In [118]:
```
a_1 = np.arange(1,11)
b_1 = np.arange(11,21)
rst1 = np.dot(a_1,b_1)
print(rst1)
print(rst1.shape)
print(a_1.shape, b_1.shape)
```

```
935
()
(10,) (10,)
```

```
The result of the dot product using np.dot is the same as hand-calculated
result.
```

In [117]:
```
rst2 = a_1.dot(b_1)
print(rst2)
print(rst2.shape)
```

```
935
()
```

 What is the shape of each of your arrays? What do you notice? **Comment below!**

```
The shapes of a_1 and b_1 are (10,), and the shape for the dot product
result is (). This means that the dot product is a 0-D array, which is a
single number.
```

---

10. Now lets see what can go wrong when even vectors have **multiple dimensions!**.

- Create a **column vector** `a_1` with the integers between `1-10` (including 10). Make sure its shape is `(10,1)`.
- Create a **column vector** `b_1` with the integers between `11-20` (including 20). Make sure its shape is `(10,1)`.

In [126]:
```
a_1 = np.arange(1,11).reshape((10,1))
b_1 = np.arange(11,21).reshape((10,1))
```

In [127]:
```
print(a_1)
print(b_1)
```

```
[[ 1]
 [ 2]
 [ 3]
 [ 4]
 [ 5]
 [ 6]
 [ 7]
 [ 8]
 [ 9]
 [10]]
[[11]
 [12]
 [13]
 [14]
 [15]
 [16]
 [17]
 [18]
 [19]
 [20]]
```

Now take their dot product:

In [128]:
```
a_1.dot(b_1)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last
t)
Input In [128], in <cell line: 1>()
----> 1 a_1.dot(b_1)

ValueError: shapes (10,1) and (10,1) not aligned: 1 (dim 1) != 10 (dim 0)
```

What do you notice? How can we fix this? **Comment below!** and then fix it in code!

Hint: Check out the `transpose` function, whose alias is `.T`

```
I have noticed that the dot product is not performed successfully, because
the shapes of a_1 and b_1 are not aligned. I added the fixed code below.
```

In [130]:
```
a_1.T.dot(b_1)
```

Out[130]:
```
array([[935]])
```