La programación grafica se fundamenta en la idea de manipular información almacenada en unas estructuras llamadas **vectores** y **matrices**(recordar este contenido de algebra I algebra II y algebra III ). La única forma de poder simular esto en python es usando listas, pero esto nos limita mucho a la hora de realizar operaciones matemáticas. **NumPy** nos viene a resolver este problema con un tipo de dato llamado array, imagino que este dato me permite hacer todas las operaciones que yo hacia con vectores y matrices en Algebra y Calculo. Un array en python se crea a partir de una lista siempre y cuando tengamos importado el módulo NumPy, con el metodo array del modulo NumPy.

*Una Dimension:*

import numpy as np

array = np.array([1, 2, 3, 4, 5])

print(array)

print(type(array))

print(array.ndim)

print(array.shape)

Output:

>>>[1 2 3 4 5]

>>><class ‘numpy.ndarray’>

>>>1

>>>(5,)

Como podemos ver hay una ligera diferencia a la hora de presentar una lista y un array, y es que estos no aparecen con sus valores separados por coma.

Estos arrays creados **a partir** de una **lista** se consideran de un array de **una** **dimensión** y también se conocen como **vectores**.

Podemos consultar las propiedades y dimensiones de un array usando los metodos **ndim**, el cual devuelve un valor **int** que corresponde al numero de dimensiones que posee el array. Otro metodo que podemos usar es el metodo **shape** el cual nos devuelve una **tupla** con tantos elementos como dimensiones tenga la tupla, en este caso ese numero 5 hace referencia a cuantos elementos tiene nuestro array.

Ahora veamos como podemos obtener un array de dos dimensiones. Esto se hace anidando dos listas dentro de la lista principal, podemos deducir de aquí que para crear arrays de mas dimensiones solo tenemos que agregar mas listas anidadas a la lista principal. Ahora veamos como se hace y que devuelven los metodos que vimos antes

*Dos Dimensiones:*

import numpy as np

array = np.array([

[1, 2, 3, 4, 5]

[6, 7, 8, 9, 10]

])

print(array)

print(type(array))

print(array.ndim)

print(array.shape)

Output:

>>>[[ 1 2 3 4 5]

[ 6 7 8 9 10]]

>>><class ‘numpy.ndarray’>

>>>2

>>>(2,5)

Aquí vemos ciertas diferencias. Primero la manera de imprimirlo por pantalla es diferente, se imprime como una matriz, o sea como una tabla. Su tipo sigue siendo el mismo, un numpy.ndarray. Por su puesto su dimensión es 2 y el mayor cambio lo vemos aquí porque la estructura de la tupla que vimos cambia. Ahora se muestra el numero de dimensiones en el primer elemento, y en el segundo el numero de elementos que tiene el array.(Podríamos decir que seria sus valores en X y Y)

Para finalizar esta parte, también podemos saber el tipo del array, o mas bien el tipo de los elementos que componen el array con el metodo siguiente:

import numpy as np

array = np.array([1, 2, 3, 4, 5])

array\_float = np.array([1, 2, 3, 4, 5.123])

array\_u4=np.array([“hola”,”que”,”tal”])

print(array.dtype)  
print(array\_float.dtype)

print(array\_u4.dtype)

Output:

>>>int32

>>>float64

>>>U4

Este **int32** hace referencia a que el array contiene enteros de 32 bits, en el caso del segundo array llamado array\_float, tenemos un **flotante** de **64** **bits**. En el ultimo caso vemos que para el array que contiene cadenas, obtenemos una **U** de **Unicode**, esto significa que el array contiene texto. Si solamente ponemos letras tendremos **U4**,de tener letras y números obtendremos **U6**.

Array Pregenerados:

Podemos generar arrays con metodos que vienen preconcebidos dentro del modulo numpy, como por ejemplo **np.zeros()**, **np.ones()**. Estos metodos vana a generar matrices según los argumentos que le pasemos. Si le pasamos un solo argumento significa que la dimensión en una, y que ese argumento es la cantidad de veces que tiene que generar los ceros. De pasarle dos argumentos, uno de ellos es la dimensión, y el otro la cantidad de veces que tiene que generar los ceros o unos de la siguiente forma.

import numpy as np

print(np.zeros(3))

print(np.zeros([3,3]))

print(np.ones([3,3]))

Output:

>>>[0 0 0]

>>>[[0 0 0]

[0 0 0]]

>>>[[1 1 1]

[1 1 1]]

Es importante recordar que en algebra habían unas matrices importantes que eran las matrices cuadradas con todo cero y su diagonal en uno. Estas matrices se pueden crear muy fácil con el metodo **eye** pasándole solamente el parámetro de cuanto quieres que sea de largo esta matriz cuadrada, de la siguiente forma:

import numpy as np

print(np.eye(3))

Output:

>>>[1 0 0]

[0 1 0]

[0 0 1]]

Estos arrays se **llaman arrays de identidad**.

Otro metodo generador en los arrays es el metodo **arange**. Si a este metodo se le da un solo argumento, este simplemente generará un array de una dimensión con un rango hasta el numero que se le pasó, incluyendo el cero. Podemos cambiar el tipo de datos que un array contiene simplemente pasándole un punto detrás del argumento, como querioendo decir que este range, deberá ser de números enteros, pero que su array.dtype es de flotantes. El arange funciona igual que el range, o sea que si sele pasa un tercer argumento, este será el salto entre números.

import numpy as np

print(np.arange(-20,20,5))

Output:

>>>[-20 -15 -10 -5 0 5 10 15]

Operaciones Básicas:

Señores esto es Algebra, se pueden hacer todas las operaciones que se pueden hacer en algebra siempre y cuando se cumplan las mismas condiciones que deben cumplirse para hacer las operaciones algebraicas pertinentes.

Suma y Resta: Mismo numero y tamaño de dimensiones

Producto, División y Potencia: numero de Columnas de la primera coincide con el numero de Filas de la segunda.

import numpy as np

#Dados dos arrays

array\_1 = np.array([1, 2, 3, 4,])

array\_2 = np.array([5 ,6, 7, 8,])

#Los Sumamos o Restamos

print(array\_1 + array\_2)

#Los Multiplicamos, Dividimos o Potenciamos

print(array\_1 \* array\_2)

Output:

>>>[6 8 10 12]

>>>[5 12 21 32]

Los arrays tambiens se pueden multiplicar por números, lo que se hace es que cada numero independiente dentro del array es multiplicado por este número. Esto seria equivalente a tener un array con un solo elemento u como el numero de columnas y filas es coincidente, se puede hacer la operación

Otra operación que podemos hacer es obtener el recíproco elevando el Aray a la menos uno como se hacia en algebra.

Aquí podemos tener una visual de como se ve una operación de matrices de dos dimensiones:

import numpy as np

#Dados dos arrays bidimensionales

array\_1 = np.array([[1, 2], [3, 4,]])

array\_2 = np.array([[5 ,6], [7, 8,]])

#Los Sumamos

print(array\_1 + array\_2)

Output:

>>>[[ 6, 8],

[10 12]])

Arrays de una Dimensión:

En los arrays como en las listas y demás colecciones de valores, podemos usar **Indices** para acceder a elementos que se encuentran en el array. También podemos hacer todas las operaciones que se pueden hacer con los índices como por ejemplo la asignación de un valor dependiendo de su índice, o sea, los arrays con mutables. También podemos usar el slicing para trabajar con un tramo del array.

Algo a tener en cuenta a la hora de trabajar con índices, y con subarrays, es que todos los cambios que hagamos en los subarrays se verán reflejados en el array original ya que al final apuntan al mismo objeto en la memoria.

De querer hacer una copia para no modificar el valor original del array deberemos usar el metodo **copy()** del módulo numpy, pasándoselo al array del que queremos hacer una copia y luego hacemos slicing.

Arrays dos una Dimensiones:

Aquí la cosa se pone un poco mas compleja. Cuando le pasamos un índice a un array de dos dimensiones, recordemos que un array de dos dimensiones no es mas de listas dentro de una lista, este nos devolverá la lista que se encuentra en esa posición, o sea, nos devolverá la fila correspondiente a ese índice. Aquí decimos que los arrays de dos dimensiones tienen dos índices uno en **X** y otro en **Y**. De igual forma para acceder a uno de los valores que están contenidos en la lista podemos simplemente volverle a pasar los corchetes y de esa forma accedemos al valor que queremos de manera normal como lo haríamos en python.

Ahora aquí es donde cambia la cosa. En el caso del slicing ya no va a ser igual que como se aplica en python. Es un tanto diferente ya que en vez que usar dos corchetes, se usa un solo corchete, son los dos campos, cada uno con su inicio y fin separados por dos puntos, y a si vez separados por una coma, definiéndose el primero como la cantidad de filas que se van a incluir y el segundo, la cantidad de columnas:

import numpy as np

#Dados dos arrays bidimensionales

array\_1 = np.rarray([[1,2,3],[4,5,6],[7,8,9]])

print(array\_1)

print(array\_1[1:,:-1])

Output:

>>>[[1 2 3]

1> [4 5 6],

[7 8 9]]

<-1

>>>[[4 5],

[7 8]])

Como podemos apreciar aquí, vemos que lo que aparece antes de la coma, define las filas que se van a imprimir, ósea se imprimen las filas **1:** ósea de la segunda hasta el final. El segundo par de valores, después de la coma, indica las columnas que se devolverán, en este caso **:-1** o lo que se interpreta como desde cero hasta la penúltima.

Fancy Index:

El Fancy index no es mas que el hecho de pasarle una lista de filas, al index normal de una matriz bidimensional que referencia mas de un índice, y así referenciamos a mas de una columna. De esta manera podemos modificar mas de una columna al mismo tiempo. Esto se ve de la siguiente manera:

import numpy as np

array\_1 = np.array([[1,2,3],[4,5,6],[7,8,9]])

print(array\_1)

print(array\_1[[0,-1]] = 99)

Output:

>>>[[1 2 3]

[4 5 6],

[7 8 9]]

>>>[[99 99 99]

[ 4 5 6],

[99 99 99]]

Aquí por ejemplo referenciamos a la primera y ultima filas, y modificamos todos sus valores.

Arrays de 3 y más Dimensiones:

Para hacer un array de tres dimensiones simplemente lo podemos hacer, o agregando un nivel más al anidar de listas, o simplemente pasándole un argumento más a nuestras funciones generadores como np.zeros(). Una matriz tridimensional esta formada por tres matrices bidimensionales, por lo que si ejecutamos el comando **print(np.zeros([3,3,3]))** este será el resultado:

[[[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. 0. 0.]]]

Como vemos tenemos tres matrices bidimensionales anidadas. Esto si lo escribimos manualmente seria algo así:

array = np.array( [ [ [0,0,0], [0,0,0], [0,0,0] ] , [ [0,0,0], [0,0,0], [0,0,0] ] , [ [0,0,0], [0,0,0], [0,0,0] ] ] )

Aquí se puede ver claramente diferenciado en colores las tres matrices de dos dimensiones cada una, que contienen a su vez tres matrices de una dimensión.

Un metodo muy interesante es el metodo **reshape()** que es un metodo que me hubiese venido bien saber antes. Este metodo puede tomar una serie de elementos en una matriz de una cierta cantidad de dimensiones y manteniendo sus valores, cambiar el numero de dimensiones. Claro, esto solo se puede hacer si la división de filas con columnas del anterior estado con el nuevo da una matriz cerrada:

import numpy as np

array\_1 = np.arange(1,10)

print(array\_1)

print(array\_1.reshape(3,3)

Output:

>>>[ 1 2 3 4 5 6 7 8 9 ]

>>>[ [ 1 2 3 ]

[ 4 5 6 ],

[ 7 8 9 ] ]

Arrays Transpuestos:

Las matrices transpuestas son aquellas matrices donde las filas de una son las columnas de otra y viceversa. Recordemos que se denotan así: **A** y **A’** para la matriz transpuesta de A. por su puesto la transpuesta de la transpuesta de A es la propia A.

Obtener la transpuesta de una matriz en python con NumPy es absurdamente fácil ya que hay un metodo en numpy que lo hace por nosotros el cual se llama “T” y se usa así:

array.T

Además del metodo ‘T’, existe otro metodo, mucho mas especifico para convertir una matriz en su transpuesta el cual se llama **swapaxes()**. Este metodo nos permite deliberadamente cambiar una dimensión por la otra. En una matriz de dos dimensiones no tiene ningún sentido ya que los ejes que serian 0 y 1 (los ejes comienzan por cero como los índices), daría igual cambiar el cero por el uno, que el uno por el cero. En cambio, cuando estamos haciendo y transpuestas de matrices de 3 dimensiones, este metodo si nos viene más útil ya que podemos decir que eje será cambiado por cual.

Funciones Universales:

**np.add** (array\_1, array\_2) : Suma dos arrays

**np.substract** (array\_1, array\_2) : Substrae el primero **del** segundo 2 – 1.

**np.sqrt** (array\_1) : devuelve un array con la raíz de cada valor

**np.power** (array\_1 , 2): toma dos argumentos, el array y la potencia. Y devuelve un array con la potencia elegida de cada valor.

**np.sign** (array\_1) : nos devuelve un array con 1 , -1 o 0 como valores dependiendo si los valores son positivos, negativos o cero.

**np.sen** (array\_1) : devuelve matriz con sen de cada valor.

**np.cos** (array\_1) : devuelve matriz con cos de cada valor.

**np.tan** (array\_1) : devuelve matriz con tan de cada valor.

**np.deg2rad** (array\_1) : devuelve matriz con ángulo en radianes de cada valor.

**np.maximum** (array\_1 , array\_2) : devuelve matriz con elemento más grande.

**np.equal** (array\_1 , array\_2) : devuelve True o False segun sean o no iguales las matrices en cada valor de una nueva matriz formada por elementos True y False.

**np.greater** (array\_1 , array\_2) : devuelve **True** o **False** segun sea **mayor** o **no** los elementos del primer array con respecto a los del segundo, cada valor de una nueva matriz formada por elementos True y False.

**np.fabs** (array\_1) : devuelve matriz con valores absolutos de cada valor.

**np.ceil** (array\_1) : devuelve matriz con valores redondeados al alza de cada valor.

**np.floor** (array\_1) : devuelve matriz con valores redondeados a la baja de cada valor.

**np.round** (array\_1) : devuelve matriz con valores redondeados de cada valor.

Funciones Aleatorias:

Se llaman funciones aleatorias y se les trata a parte porque tienen su propio modulo que por supuesto se llama **random**.

**np.random.rand()** : Devuelve un elemento entre cero y uno

**np.random.rand(4)** : Devuelve 4 elementos entre cero y uno

**np.random.rand(4, 2)** : Devuelve un array de dos dimensiones de 4 elementos entre cero y uno

**np.random.uniform( 10, size = [2, 2, 2])** : Aqui lo que hacemos es que el random valla hasta el numero 10, (también podríamos pasarle otro parámetro ya que tiene dos parámetros de los cuales el primero está default a cero, así que para cambiarlo solamente hay que agregar otro elemento entre el primero y el tamaño “size”) y por supuesto, size hace referencia a las dimensiones de la matriz.

***Si en vez de querer generar números flotantes queremos numero enteros usamos:***

**np.random.randint(-10, 10, size = [2, 2, 2])**

Funciones de permutaciones:

**np.random.shuffle(array\_1)** : Devuelve la misma matriz desordenada.

**np.random.permutation(10)** : Devuelve una matriz con elementos de un rango desordenados.

Filtrado de Arrays:

**np.unique(array\_1) :** Es como un set a una lista, elimina duplicados.

**np.in1d( [elementos a comprobar], array\_1) :** Recibe una lista de elementos a comprobar si están o no, y el array a comprobar y devuelve un array de True o False, dependiendo si están o no.

**np.where(array\_1 < 0, 0, array\_1)**

El **where** nos permite modificar los valores de una matriz pasándole una **condición** como **primer** **argumento** que se compruebe con los valores, si la condición es verdadera, establecerá el **numero** en el **segundo** **argumento**, en el **array** que toca como **tercer** **argumento.**

Otro metodo muy útil es el metodo **all()**, el cual comprueba que todos los elementos dentro e un array compuesto por elementos booleanos , es completamente True, devolviendo un solo True o un solo False.

El metodo **any()** devuelve True si al menos uno de los elementos dentro de la matriz es True.

**IMPORTANTE**: Sobre estos dos metodos, ellos reciben argumentos si la matriz es de más de una dimensión, este argumento es el eje que recordemos que comienza por cero.

Metodos Internos:

Esos metodos se usan sobre los arrays como tal, ya que son metodos internos del propio modulo.

**array\_1.sum()** : Devuelve la suma de todos los elementos de la matriz, int o float.

**array\_1.mean()** : Devuelve la media de todos los elementos de la matriz, int o float.

**array\_1.std()** : Devuelve la desviación estándar de todos los elementos de la matriz, int o float.

**array\_1.var()** : Devuelve la varianza de todos los elementos de la matriz, int o float.

**array\_1.sort()** : Devuelve las filas y las columnas ordenadas de mayor a menor. Este metodo puede tomar un argumento del tipo eje (0,1,2).

Arrays y Persistencia:

Persistencia significa que podemos guardar nuestros arrays en ficheros para su posterior uso. Los arrays se pueden guardar básicamente de dos formas, binaria y en formato de texto. Estas dos formas de escribir arrays en un fichero tienen sus ventajas y desventajas, en forma binaria no se puede abrir en el editor de texto, pero podemos guardar mas de un array, mientras que en formato de texto solamente podemos guardar un array, pero podemos leerlo en un editor de texto. También cabe destacar que el formato del archivo en el que se guardan los arrays del módulo numpy es .**npy**.

Metodos:

**BINARIO >>>**

**np.save( ‘array\_1.npy’ , array\_1 )**

Luego para abrirlo podemos usar el metodo load() e igualarlo a una variable para que quede guardado en la memoria.

**array\_1 = np.load( ‘array\_1.npy’ )**

Si queremos salvar más de un array en un archivo binario usamos un metodo diferente y además el tipo de archivo también cambia a **.npz**, justo después tenemos que especificar una clave para cada array:

**np.savez( ‘arrays.npz’ , arr1 = array\_1 , arr2 = array\_2 , arr3 = array\_3 )**

Cuando cargamos esto, lo que nos devuelve del archivo binario es un objeto del tipo:

<numpy.lib.npyio.NpzFile at 0x27554abc6d8>

Esto sucede poque lo que tenemos es un objeto que pertenece a esa clase, el cual contiene un diccionario con las claves que le pasamos al guardarlo, como claves, y los arrays como valores y no lo puede leer, así como así. Recordemos que los objetos necesitan metodos para poderse imprimir por pantalla como el metodo \_\_str\_\_ o el metodo print(). En el caso de un diccionario recordemos que accedemos a el con sus claves por tanto y demás, si queremos acceder simplemente hacemos esto:

**arrays = np.load( ‘arrays.npz’ )**

**arrays[‘arr1’]**

**TEXTO >>>**

**np.savetxt( ‘array\_1.txt’ , array\_1 )**