**4. Dimensiones**

scalar: dim = 0 Un solo dato o valor

vector: dim = 1 Listas de Python

matriz: dim = 2 Hoja de cálculo

tensor: dim > 3 Series de tiempo o Imágenes

Declarando un escalar.

scalar = np.array(42)

print(scalar) ----> 42

scalar.ndim ----> 0

Declarando un vector.

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

print(vector) ----> [1 2 3]

vector.ndim ----> 1

Declarando una matriz.

matriz = np.array([[1, 2, 3], [4, 5, 6]])

print(matriz)

matriz.ndim

---->[[1 2 3]

[4 5 6]]

----> 2

Declarando un tensor.

tensor = np.array([[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]],[[13, 13, 15], [16, 17, 18], [19, 20, 21], [22, 23, 24]]])

print(tensor)

tensor.ndim

----> [[[ 1 2 3]

[ 4 5 6]

[ 7 8 9]

[10 11 12]]

[[13 13 15]

[16 17 18]

[19 20 21]

[22 23 24]]]

----> 3

Agregar o eliminar dimensiones

Se puede definir el número de dimensiones desde la declaración del array

vector = np.array([1, 2, 3], ndmin = 10)

print(vector) ----> [[[[[[[[[[1 2 3]]]]]]]]]]

vector.ndim ----> 10

Se pueden expandir dimensiones a los array ya existentes.

Axis = 0 hace refencia a las filas, mientras que axis = 1 a las columnas.

expand = np.expand\_dims(np.array([1, 2, 3]), axis = 0)

print(expand) ----> [[1 2 3]]

expand.ndim ----> 2

Remover/comprimir las dimensiones que no están siendo usadas.

print(vector, vector.ndim) ----> [[[[[[[[[[1 2 3]]]]]]]]]] 10

vector\_2 = np.squeeze(vector)

print(vector\_2, vector\_2.ndim) ----> [1 2 3] 1

5. 🔀**Creando arrays**

Este métodode NumPy nos permite generar arrays sin definir previamente una lista.

np.arange(0,10)

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

Un tercer argumento permite definir un tamaño de paso.

np.arange(0,20,2)

----> array([ 0, 2, 4, 6, 8, 10, 12, 14, 16, 18])

np.zeros() nos permite definir estructuras o esquemas.

np.zeros(3)

----> array([0., 0., 0.])

np.zeros((10,5))

----> 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., 0., 0., 0.],

[0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0.]])

De iugal forma tenemos np.ones()

np.ones(3)

----> array([1., 1., 1.])

np.linspace() permite generar una arrary definiendo un incio, un final y cuantas divisiones tendrá.

np.linspace(0, 10 , 10)

----> array([ 0.,1.11111111,2.22222222, 3.33333333, 4.44444444,

5.55555556, 6.66666667, 7.77777778, 8.88888889, 10.])

También podemos crear una matriz con una diagonal de 1 y el resto de 9.

np.eye(4)

----> array([[1., 0., 0., 0.],

[0., 1., 0., 0.],

[0., 0., 1., 0.],

[0., 0., 0., 1.]])

Otra metódo importante es generar números aleatorios.

np.random.rand()

----> 0.37185218178880153

También se pueden generar vectores.

np.random.rand(4)

----> array([0.77923054, 0.90495575, 0.12949965, 0.55974303])

Y a su vez generar matrices.

np.random.rand(4,4)

----> array([[0.26920153, 0.24873544, 0.02278515, 0.08250538],

[0.16755087, 0.59570639, 0.83604996, 0.57717126],

[0.00161574, 0.27857138, 0.33982786, 0.19693596],

[0.69474123, 0.01208492, 0.38613157, 0.609117 ]])

NumPy nos permite tambien generar números enteros.

En este caso números enteros entre el 1 y 14

np.random.randint(1,15)

----> 4

También podemos llevarlos a una estructura definida.

np.random.randint(1,15, (3,3))

----> array([[ 8, 2, 6],

[ 7, 1, 8],

[11, 14, 4]])

6. 🟦🔹**Shape y Reshape**

Shape me indica la forma del arreglo

Reshape transforma el arreglo mientras se mantengan los elementos.

arr = np.random.randint(1,10,(3,2))

arr.shape

----> (3, 2)

arr

----> array([[4, 2],

[4, 8],

[4, 3]])

arr.reshape(1,6)

----> array([[4, 2, 4, 8, 4, 3]])

arr.reshape(2,3)

----> array([[4, 2, 4],

[8, 4, 3]])

np.reshape(arr,(1,6))

----> array([[4, 2, 4, 8, 4, 3]])

Se puede hacer un reshape como lo haría C.

np.reshape(arr,(2,3), 'C')

----> array([[4, 2, 4],

[8, 4, 3]])

También se puede hacer reshape a como lo haría Fortran.

np.reshape(arr,(2,3), 'F')

----> array([[4, 4, 8],

[4, 2, 3]])

Además existe la opción de hacer reshape según como esté optimizado nuestro computador. En este caso es como en C.

np.reshape(arr,(2,3), 'A')

----> array([[4, 2, 4],

[8, 4, 3]])

7. 🧮**Funciones principales de NumPy**

arr = np.random.randint(1, 20, 10)

matriz = arr.reshape(2,5)

matriz

----> array([[18, 8, 3, 11, 1],

[15, 10, 13, 9, 15]])

arr

----> array([18, 8, 3, 11, 1, 15, 10, 13, 9, 15])

arr.max() ----> 18

matriz.max() ----> 18

Podemos regresar los máximos de cada fila o columna especificando el eje.

matriz.max(1) ----> array([18, 15])

matriz.max(0) ---->array([18, 10, 13, 11, 15])

Tambien tenemos .argmax() que nos devuelve la posición del elemento

arr.argmax() ----> 0

matriz.argmax(0) ----> array([0, 1, 1, 0, 1])

De forma análoga tenemos .min()

arr.min() ----> 1

arr.argmin() ----> 4

matriz.min(0) ----> array([15, 8, 3, 9, 1])

matriz.argmin(1) ----> array([4, 3])

Podemos saber la diferencia de valor más bajo con el más alto.

arr.ptp() # 18 - 1 ----> 17

matriz.ptp(0) ----> array([ 3, 2, 10, 2, 14])

Para hacer análisis estádistico se tienen la siguientes funciones.

Ordenar los elementos:

arr.sort() ----> array([1, 3, 8, 9,10, 11,13, 15,15,18])

Obtener un percentil:

np.percentile(arr, 0) ----> 1.0

Mediana:

np.median(arr) ----> 10.5

Desviación estándar:

np.std(arr) ----> 5.080354

Varianza:

np.var(arr) ----> 25.81000

Promedio

np.mean(arr) ----> 10.3

Lo mismo aplica para las matrices.

np.median(matriz, 1) ----> array([ 8., 15.])

Se pueden unir dos arrays por medio de la concatenación

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

b= np.array([5, 6])

np.concatenate((a,b), axis = 0)

----------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-213-97c6fb2c6837> in <module>()

----> 1 np.concatenate((a,b), axis = 0)

<\_\_array\_function\_\_ internals> in concatenate(\*args, \*\*kwargs)

ValueError: all the input arrays must have same number of dimensions, but the array at index 0 has 2 dimension(s) and the array at index 1 has 1 dimension(s)

El error anterior es debido a a tiene 2 dimensiones mientras que b tiene 1.

print(a.ndim) ----> 2

b.ndim ----> 1

b = np.expand\_dims(b, axis = 0)

np.concatenate((a,b), axis = 0)

----> array([[1, 2],

[3, 4],

[5, 6]])

De igual forma podemos agregarlo en el otro eje

np.concatenate((a,b), axis = 1)

-----------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-217-3ae7237876ab> in <module>()

1 # De igual forma podemos agregarlo en el otro eje

----> 2 np.concatenate((a,b), axis = 1)

<\_\_array\_function\_\_ internals> in concatenate(\*args, \*\*kwargs)

ValueError: all the input array dimensions for the concatenation axis must match exactly, but along dimension 0, the array at index 0 has size 2 and the array at index 1 has size 1

Como b es una fila y no una columna, no se puede concatenar a menos que se aplique la transpuesta.

np.concatenate((a,b.T), axis = 1)

----> array([[1, 2, 5],

[3, 4, 6]])

**8. Copy**

.copy() nos permite copiar un array de NumPy en otra variable de tal forma que al modificar el nuevo array los cambios no se vean reflejados en array original.

arr = np.arange(0, 11)

arr ----> array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

arr[0:6] ----> array([0, 1, 2, 3, 4, 5])

trozo\_de\_arr = arr[0:6]

trozo\_de\_arr[:] = 0

trozo\_de\_arr ----> array([0, 0, 0, 0, 0, 0])

Se han modificado los datos del array original porque seguía haciendo referencia a esa variable.

arr ----> array([ 0, 0, 0, 0, 0, 0, 6, 7, 8, 9, 10])

arr\_copy = arr.copy()

arr\_copy[:] = 100

arr\_copy ----> array([100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100])

arr ----> array([ 0, 0, 0, 0, 0, 0, 6, 7, 8, 9, 10])

9. Condiciones

Las condiciones nos permiten hacer consultas más específicas.

arr = np.linspace(1,10,10, dtype = 'int8')

arr ----> array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10], dtype=int8)

Regresa un array de booleanos donde la condiciones se cumple.

indices\_cond = arr > 5

indices\_cond

----> array([False, False, False, False, False, True, True, True, True, True])

Regresa los valores para donde la condiciones True.

arr[indices\_cond] ----> array([ 6, 7, 8, 9, 10], dtype=int8)

Se pueden agregar múltiples condiciones.

arr[(arr > 5) & (arr < 9)] ----> array([6, 7, 8], dtype=int8)

De igual forma modificar los valores que cumplan la condición.

arr[arr > 5] = 99

arr ----> array([ 1, 2, 3, 4, 5, 99, 99, 99, 99, 99], dtype=int8)

**10 Operaciones**

Existen diferentes operaciones que se pueden usar para los arrays de NumPy.

lista = [1,2]

lista ----> [1, 2]

Una lista de Python entiende que quieres duplicar los datos. Cosa que no buscamos.

lista \* 2 ----> [1, 2, 1, 2]

arr = np.arange(0,10)

arr2 = arr.copy()

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

Ahora multiplicamos por un escalar:

arr \* 2 ----> array([ 0, 2, 4, 6, 8, 10, 12, 14, 16, 18])

Operación suma de escalar:

arr + 2 ----> array([ 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

División con un escalar

Como en este caso la primera posición del array es 0, muestra un error pero no detiene el proceso.

1/arr

/usr/local/lib/python3.7/dist-packages/ipykernel\_launcher.py:1: RuntimeWarning: divide by zero encountered in true\_divide

"""Entry point for launching an IPython kernel.

array([ inf, 1. , 0.5 , 0.33333333, 0.25 ,

0.2 , 0.16666667, 0.14285714, 0.125 , 0.11111111])

Elevar a un escalar:

arr\*\*2 ----> array([ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81])

Sumar dos arrays de igual dimensiones las hace elemento por elemento:

arr + arr2 ----> array([ 0, 2, 4, 6, 8, 10, 12, 14, 16, 18])

Lo mismo aplica para matrices.

matriz = arr.reshape(2,5)

matriz2 = matriz.copy()

matriz

----> array([[0, 1, 2, 3, 4],

[5, 6, 7, 8, 9]])

matriz - matriz2

----> array([[0, 0, 0, 0, 0],

[0, 0, 0, 0, 0]])

Una operación importante es la de punto por punto, aquí dos formas de hacerla:

np.matmul(matriz, matriz2.T)

----> array([[ 30, 80],

[ 80, 255]])

matriz @ matriz2.T

----> array([[ 30, 80],

[ 80, 255]])

<https://pandas.pydata.org/docs/user_guide/10min.html> Funciones panda

<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html>

Comparación SQL y Pandas

