El futuro digital es de todos

MinTIC

Numpy

OPERADO POR:

Universidad Tecnológica de Pereira

Mision TIC 2022

RUTA DE APRENDIZAJE 1

Numpy

Numpy es una librería para la ciencia de los datos en Python. Proporciona un objeto matriz multidimensional de alto rendimiento y herramientas para trabajar con estas matrices. NumPy también permite a los desarrolladores de Python realizar de forma rápida una amplia variedad de cálculo numéricos.

https://numpy.org/

Matrices

Una matriz numpy es una cuadrícula de valores, todos del mismo tipo, y está indexada por una tupla de enteros no negativos. El número de dimensiones es el rango de la matriz; la forma de una matriz es una tupla de números enteros que dan el tamaño de la matriz a lo largo de cada dimensión.

Podemos inicializar matrices numpy desde listas de Python anidadas y acceder a elementos usando corchetes:

```
In [13]: from numpy import random as r
```

```
In [2]: print(r.choice(['Andres','Juan','Pedro', 'Mateo'], size= r.choice([1,2],p=[0.1,0.9]) , p=[0.5,0.2,0.2,0.1], replace=False))
['Mateo' 'Andres']
```

```
In [3]: import numpy as np

        a = np.array([34, 25, 7])   # Crear una matriz de rango 1
```

```
In [4]: print(type(a))           # Prints "<class 'numpy.ndarray'>"
<class 'numpy.ndarray'>
```

```
In [5]: print(a.shape)           # Prints "(3,)"
(3,)
```

```
In [6]: print(a[0], a[1], a[2])   # Prints "1 2 3"
34 25 7
```

```
In [7]: a[0] = 5                    # Cambiar un elemento de la matriz

In [8]: print(a)                    # Prints "[5, 2, 3]"
[ 5 25  7]

In [9]: b = np.array([[1,2,3],[4,5,6]])    # Crear una matriz de rango 2

In [10]: print(b.shape)                     # Prints "(2, 3)"
(2, 3)

In [11]: print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
1 2 4
```

NumPy también proporciona muchas funciones para crear matrices y así hacer nuestro proceso mas fácil de manejar:

In [137]:
```
import numpy as np

matriz = np.zeros((3,3))   # Crear una matriz de todos los ceros
print(matriz)              # Prints "[[ 0.  0.]
                           #         [ 0.  0.]]"
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

In [138]:
```
b = np.ones((1,2))    # Crear un conjunto de todos ellos
print(b)              # Prints "[[ 1.  1.]]"
```

```
[[1. 1.]]
```

In [143]:
```
c = np.full((2,2), 7)  # Crear una matriz constante
print(c)               # Prints "[[ 7.  7.]
                       #         [ 7.  7.]]"
```

```
[[7 7]
 [7 7]]
```

In [13]:
```
d = np.eye(2)          # Crear una matriz de identidad 2x2
print(d)               # Prints "[[ 1.  0.]
                       #          [ 0.  1.]]"
```

```
[[1. 0.]
 [0. 1.]]
```

In [5]:
```
e = np.random.random((2,2))  # Crear una matriz llena de valores aleatorios
print(e)                     # Podría imprimir "[[ 0.91940167  0.08143941]
                             #                  [ 0.68744134  0.87236687]]"
```

```
[[0.00175892 0.12706911]
 [0.03101381 0.71757546]]
```

El manejo de **matrices** es muy importante para nosotros en programación, permite manejar de una forma mas eficiente los datos de nuestro problema y ubicar de igual manera la información que necesitamos para nuestros **cálculos o búsquedas**.

Indexación de matrices

Numpy ofrece varias formas de indexar en matrices.

Rebanar

Similar a las listas de Python, las matrices **NumPy** se pueden cortar. Dado que las matrices pueden ser multidimensionales, debe especificar un segmento para cada dimensión de la matriz:

In [165]:
```
import numpy as np

# Crear la siguiente matriz de rango 2 con forma (3, 4)
# [[ 1  2  3]
#  [ 5  6  7]
#  [ 9 10 11 ]]
a = np.array([[1,2,3], [5,6,7], [9,10,11]])
```

In [148]:
```
# Usar el rebanado para sacar el subconjunto que consiste en las 2 primeras filas
# y las columnas 1 y 2; b es el siguiente conjunto de forma (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
print(b)
```

```
[[2 3]
 [6 7]]
```

In [167]:
```
print(np.fliplr(a))
```

```
[[ 3  2  1]
 [ 7  6  5]
 [11 10  9]]
```

In [149]:
```
# Una rebanada de una matriz es una vista en los mismos datos, por lo que modificarla
# modificará la matriz original.
print(a[0, 1])   # Prints "2"
```

```
2
```

In [150]:
```
b[0, 0] = 77     # b[0, 0] es la misma pieza de datos que a[0, 1]
```

In [151]:
```
print(a[0, 1])   # Prints "77"
```

```
77
```

También puede mezclar la indexación de **enteros** con la indexación de sectores. Sin embargo, al hacerlo, se obtendrá una matriz de rango más bajo que la matriz original.

In [21]:
```
import numpy as np

# Crear la siguiente matriz de rango 2 con forma (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
```

In [ ]:
```
# Dos formas de acceder a los datos de la fila del medio de la matriz.
# Mezclando la indexación de enteros con rebanadas se obtiene una matriz de rango inferior,
# mientras que usando sólo rebanadas se obtiene un conjunto del mismo rango que el
# La matriz original:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
```

In [22]:
```
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"
```

```
[5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
```

In [23]:
```
# Podemos hacer la misma distinción al acceder a las columnas de una matriz:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #         [ 6]
                             #         [10]] (3, 1)"
```

```
[ 2  6 10] (3,)
[[ 2]
 [ 6]
 [10]] (3, 1)
```

Indexación de matrices de enteros:

Cuando indexa matrices de números utilizando la división, la vista de matriz resultante siempre será una submatriz de la matriz original. Por el contrario, la indexación de matrices de enteros le permite construir matrices arbitrarias utilizando los datos de otra matriz. Aquí hay un ejemplo:

In [6]:
```
import numpy as np

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

In [8]:
```
# Un ejemplo de indexación de arreglos enteros.
# La matriz devuelta tendrá forma (3,2) y
print(a.shape)
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"
```

```
(3, 2)
[1 4 5]
```

In [26]:
```
# El ejemplo anterior de indexación de arreglos enteros es equivalente a esto:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"
```

```
[1 4 5]
```

In [27]:
```
# Cuando se usa la indexación de arreglos enteros, se puede reutilizar el mismo
# elemento de la matriz de la fuente:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"
```

```
[2 2]
```

In [28]:
```
# Equivalente al ejemplo anterior de indexación de arreglos enteros
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

```
[2 2]
```

Un truco útil con la indexación de matrices enteras es seleccionar o **mutar** un elemento de cada fila de una **matriz**.

Podemos modificar las **matrices** con la estructura que necesitemos para nuestro problema y ser mas eficientes en el proceso de solución.

In [29]:
```
import numpy as np

# Crear una nueva matriz de la cual seleccionaremos elementos
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a)  # prints "array([[ 1,  2,  3],
          #                [ 4,  5,  6],
          #                [ 7,  8,  9],
          #                [10, 11, 12]])"
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

In [31]:
```
# Crear una serie de índices
b = np.array([0, 2, 0, 1])

# Selecciona un elemento de cada fila de a usando los índices de b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
```

```
[11 16 17 21]
```

In [32]:
```
# Muta un elemento de cada fila de a usando los índices de b
a[np.arange(4), b] += 10

print(a)  # prints "array([[11,  2,  3],
          #                [ 4,  5, 16],
          #                [17,  8,  9],
          #                [10, 21, 12]])
```

```
[[21  2  3]
 [ 4  5 26]
 [27  8  9]
 [10 31 12]]
```

OPERADO POR:

Universidad Tecnológica de Pereira

Mision TIC 2022

Indexación de matriz booleana:

La indexación de matriz booleana le permite seleccionar elementos arbitrarios de una matriz. Con frecuencia, este tipo de indexación se utiliza para seleccionar los elementos de una matriz que satisfacen alguna condición. Aquí hay un ejemplo:

In [33]:
```
import numpy as np

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

In [34]:
```
bool_idx = (a > 2)  # Encuentra los elementos de a que son más grandes que 2;
                    # esto devuelve un conjunto numérico de Booleans de la misma
                    # forma que a, donde cada ranura de bool_idx dice
                    # si ese elemento de a es > 2.
```

In [35]:
```
print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"
```

```
[[False False]
 [ True  True]
 [ True  True]]
```

In [36]:
```
# Usamos la indexación de arreglos booleanos para construir un arreglo de rango 1
# que consiste en los elementos de un correspondiente a los valores Verdaderos
# de bool_idx

print(a[bool_idx])  # Prints "[3 4 5 6]"
```

```
[3 4 5 6]
```

In [37]:
```
# Podemos hacer todo lo anterior en una sola declaración concisa:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

```
[3 4 5 6]
```

Por brevedad, hemos omitido muchos detalles sobre la indexación de matrices NumpPy; si quieres saber más podemos revisar la documentación.

https://numpy.org/doc/stable/reference/arrays.indexing.html

Tipos de datos:

Cada matriz NumPy es una cuadrícula de elementos del mismo tipo. Numpy proporciona un gran conjunto de tipos de datos numéricos que puede utilizar para construir matrices. Numpy intenta adivinar un tipo de datos cuando crea una matriz, pero las funciones que construyen matrices generalmente también incluyen un argumento opcional para especificar explícitamente el tipo de datos. Aquí hay un ejemplo:

In [38]:
```
import numpy as np

x = np.array([1, 2])   # Dejar que numpy elija el tipo de datos
print(x.dtype)         # Prints "int64"
```

```
int32
```

In [39]:
```
x = np.array([1.0, 2.0])   # Dejar que numpy elija el tipo de datos
print(x.dtype)             # Prints "float64"
```

```
float64
```

In [40]:
```
x = np.array([1, 2], dtype=np.int64)   # Forzar un tipo de datos en particular
print(x.dtype)                         # Prints "int64"
```

```
int64
```

Matemáticas de matriz:

Las funciones matemáticas básicas operan por elementos en matrices y están disponibles como sobrecargas de operador y como funciones en el módulo **NumPy**.

Podemos realizar cualquier operación entre matrices, sin embargo debemos tener en cuenta la forma correcta de realizar estos cálculos.

In [41]:
```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)
```

In [42]:
```
# Suma de elementos; ambos producen la matriz
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

In [43]:
```
# Diferencia de elementos (resta); ambos producen la matriz
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

Todas las operaciones entre valores, o elementos, podemos realizarlas con las matrices utilizando la sintaxis requerida.

In [44]:
```
# Producto de elementos; ambos producen la matriz
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

In [46]:
```
# División de elemetos; ambos producen la matriz
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

In [47]:
```
# Raíz cuadrada de elemtos; produce la matriz
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Debemos tener en cuenta que **es una multiplicación por elementos, no una multiplicación de matrices**. Por esta razón debemos utilizar la dotfunción para calcular productos internos de vectores utilizando la teoría del producto punto, para multiplicar un vector por una matriz y para multiplicar matrices. dot está disponible como función en el módulo NumPy y como método de instancia de objetos de matriz:

```
In [48]: import numpy as np

         x = np.array([[1,2],[3,4]])
         y = np.array([[5,6],[7,8]])
```

```
In [49]: v = np.array([9,10])
         w = np.array([11, 12])
```

```
In [50]: # Producto interno de los vectores; ambos producen 219
         print(v.dot(w))
         print(np.dot(v, w))
```

```
219
219
```

```
In [51]: # Matriz / producto vectorial; ambos producen la matriz de rango 1 [29 67]
         print(x.dot(v))
         print(np.dot(x, v))
```

```
[29 67]
[29 67]
```

```
In [52]: # Matriz / producto de la matriz; ambos producen la matriz de rango 2
         # [[19 22]
         #  [43 50]]
         print(x.dot(y))
         print(np.dot(x, y))
```

```
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
```

Numpy proporciona muchas funciones útiles para realizar cálculos en matrices; uno de los más útiles es sum:

```
In [53]: import numpy as np

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

```
In [54]: print(np.sum(x))  # Calcular la suma de todos los elementos; imprime "10"
         print(np.sum(x, axis=0))  # Calcula la suma de cada columna; imprime "[4 6]"
         print(np.sum(x, axis=1))  # Calcula la suma de cada fila; imprime "[3 7]"
```

```
10
[4 6]
[3 7]
```

Además de calcular funciones matemáticas utilizando matrices, con frecuencia necesitamos remodelar o manipular datos en matrices. El ejemplo más simple de este tipo de operación es la transposición de una matriz; para transponer una matriz, simplemente use el **T** atributo de un objeto de matriz:

In [55]:
```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #          [3 4]]"
print(x.T)  # Prints "[[1 3]
            #          [2 4]]"
```

```
[[1 2]
 [3 4]]
[[1 3]
 [2 4]]
```

In [56]:
```
# Note que tomar la transposición de una matriz de rango 1 no hace nada:
v = np.array([1,2,3])
print(v)    # Prints "[1 2 3]"
print(v.T)  # Prints "[1 2 3]"
```

```
[1 2 3]
[1 2 3]
```

Broadcasting

El Broadcasting es un mecanismo poderoso que permite a numpy trabajar con matrices de diferentes formas al realizar operaciones aritméticas.

Con frecuencia tenemos una matriz más pequeña y una matriz más grande, y queremos usar la matriz más pequeña varias veces para realizar alguna operación en la matriz más grande.

Por ejemplo, suponga que queremos agregar un vector constante a cada fila de una matriz. Podríamos hacerlo así:

In [14]:
```
import numpy as np

# Añadiremos el vector v a cada fila de la matriz x,
# almacenando el resultado en la matriz y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Crear una matriz vacía con la misma forma que x

# Agrega el vector v a cada fila de la matriz x con un bucle explícito
for i in range(4):
    y[i, :] = x[i, :] + v
```

In [15]:
```
# Ahora y es lo siguiente
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Esto funciona; sin embargo, cuando la matriz x es muy grande, calcular un bucle explícito en Python podría ser lento. Tenga en cuenta que agregar el vector va cada fila de la matriz x es equivalente a formar una matriz vv apilando múltiples copias de v verticalmente, luego realizando la suma de elementos de x y vv. Podríamos implementar este enfoque de esta manera:

In [59]:
```
import numpy as np

# Añadiremos el vector v a cada fila de la matriz x,
# almacenando el resultado en la matriz y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Amontonar 4 copias de V una encima de la otra
print(vv)                 # Prints "[[1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]]"
y = x + vv  # Agrega x y vv elementalmente
print(y)  # Prints "[[ 2  2  4
          #         [ 5  5  7]
          #         [ 8  8 10]
          #         [11 11 13]]"
```

```
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

El Broadcasting Numpy nos permite realizar este cálculo sin crear realmente múltiples copias de v. Considere esta versión, usando transmisión:

In [15]:
```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([[1, 0],[3, 0]])
print(v)
print(3 in np.sum(v,axis=0))
#y = x + v  # Añada v a cada fila de x utilizando la radiodifusión
#print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

```
[[1 0]
 [3 0]]
False
```

La línea y = x + v funciona aunque x tiene forma (4, 3) y v tiene forma (3,) debido a la transmisión; esta línea funciona como si v realmente tuviera forma (4, 3), donde cada fila era una copia de v, y la suma se realizó por elementos.

El Broadcasting de dos matrices juntas sigue estas reglas:

1. Si las matrices no tienen el mismo rango, anteponga 1 a la forma de la matriz de rango inferior hasta que ambas formas tengan la misma longitud.

2. Se dice que las dos matrices son compatibles en una dimensión si tienen el mismo tamaño en la dimensión, o si una de las matrices tiene el tamaño 1 en esa dimensión.

3. Los arreglos se pueden transmitir juntos si son compatibles en todas las dimensiones.

4. Después de la transmisión, cada matriz se comporta como si tuviese una forma igual al máximo de formas de las dos matrices de entrada.

5. En cualquier dimensión donde una matriz tiene un tamaño 1 y la otra matriz tiene un tamaño mayor que 1, la primera matriz se comporta como si se hubiera copiado a lo largo de esa dimensión.

Estas son algunas aplicaciones del Broadcasting:

In [61]:
```
import numpy as np

# Calcular el producto exterior de los vectores
v = np.array([1,2,3]) # v tiene forma (3,)
w = np.array([4,5]) # w tiene forma (2,)
# Para calcular un producto exterior, primero reformamos v para que sea una columna
# vector de forma (3, 1); podemos entonces emitirlo contra w para rendir
# una salida de la forma (3, 2), que es el producto exterior de v y w:
# [[ 4 5]
# [ 8 10]
# [12 15]]
print(np.reshape(v, (3, 1)) * w)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
```

In [62]:
```
# Agregar un vector a cada fila de una matriz
x = np.array([[1,2,3], [4,5,6]])
# x tiene forma (2, 3) y v tiene forma (3,) por lo que transmiten a (2, 3),
# dando la siguiente matriz:
# [[2 4 6]
# [5 7 9]]
print(x + v)
```

```
[[2 4 6]
 [5 7 9]]
```

In [63]:
```
# Agregar un vector a cada columna de una matriz
# La x tiene forma (2, 3) y la w tiene forma (2,).
# Si transponemos x entonces tiene forma (3, 2) y puede ser difundida
# contra w para obtener el resultado de la forma (3, 2); transponiendo este resultado
# produce el resultado final de la forma (2, 3) que es la matriz x con
# el vector w añadido a cada columna. Da la siguiente matriz:
# [[ 5 6 7]
# [ 9 10 11]]
print((x.T + w).T)
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

In [64]:
```
# Otra solución es remodelar la w para que sea un vector de forma de la columna (2, 1);
# podemos entonces emitirlo directamente contra x para producir la misma
# salida.
print(x + np.reshape(w, (2, 1)))
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

In [65]:
```
# Multiplica una matriz por una constante:
# x tiene forma (2, 3). Numpy trata los escalares como matrices de forma ();
# estos pueden ser emitidos juntos a la forma (2, 3), produciendo el
# Siguiendo la matriz:
# [[ 2 4 6]
# [ 8 10 12]]
print(x * 2)
```

```
[[ 2  4  6]
 [ 8 10 12]]
```

El Broadcasting suele hacer que su código sea más conciso y rápido, por lo que debe esforzarse por utilizarlo siempre que sea posible.

El futuro digital es de todos

MinTIC

GRACIAS

OPERADO POR:

Universidad Tecnológica de Pereira

MisiónTIC2022