AWAKELAB

BASECAMP

Ciencia de Datos

Obtención y Preparación de Datos

Objetivo de la jornada

- Manipular datos utilizando estructuras de vectores y matrices acorde a biblioteca Numpy para resolver un problema.

La Librería Numpy

Para comenzar este apartado se dará un pequeño ejemplo a modo de explicación.

Sumaremos los elementos de una lista que contendrá los primeros mil números enteros positivos (que construiremos con la función **range**()), midiendo el tiempo que requiere la mencionada operación a través del comando mágico %**timeit**.

```
test_list = list(range(1001))
%timeit sum(test_list)

#  5 µs ± 1.26 µs per loop (mean ± std. dev. of 7 runs, 100000
loops each)
```

Ahora repitamos la misma operación empleando funciones de la librería ***NumPy***. Para ello:

- Importaremos el módulo siguiendo la convención establecida, es decir, escribiendo **import numpy as np**,

- Creamos una réplica de la anterior lista (que, en breve, empezaremos a llamar array) utilizando la función **arange**(), y
- Sumaremos sus elementos mediante la función **sum**().

```
import numpy as np

test_array = np.arange(1001))
%timeit sum(test_array)

#  94.1 µs ± 1.18 µs per loop (mean ± std. dev. of 7 runs,
10000 loops each)
```

Hemos reducido a la mitad el tiempo que precisa el sistema para realizar el cálculo numérico requerido. No obstante, quizá no logre impactarnos el hecho de pasar de 11 a 5 segundos. Intentemos forzar un tanto el anterior ejemplo incrementando de manera significativa el número de elementos.

```
n = 1000001
test_list = list(range(n))   # Python
test_array = np.arange(n)    # NumPy
%timeit sum(test_list)

# 4.77 ms ± 97.8 µs per loop (mean ± std. dev. of 7 runs, 100
loops each)


%timeit sum(test_array)

# 98.3 ms ± 10.4 ms per loop (mean ± std. dev. of 7 runs, 10
loops each)
```

El resultado ahora sí que debería ser una buena justificación de cara a decidir si invertir o no nuestro preciado tiempo en aprender a utilizar la librería *NumPy*. Esta considerable mejora en el tiempo de ejecución para cálculos numéricos se produce en cualquier tipo de operación matemática que llevemos a cabo empleando funciones de dicho módulo.

Si tenemos en mente utilizar el lenguaje de programación *Python* para analizar datos, considerando que al final casi todo se reduce a realizar cálculos numéricos con matrices de dimensiones considerables, *NumPy* se convierte entonces en una herramienta esencial.

Sin entrar en demasiados detalles técnicos, esta situación se produce debido a que:

1. Es distinta, y mucho más eficiente, la manera en que se accede a los elementos de un array de NumPy con respecto a cómo Python procede a realizar tal tarea en sus estructuras de datos básicas,
2. El número de comprobaciones intermedias a la hora de llevar a cabo cálculos numéricos es menor en NumPy, y
3. NumPy está escrito utilizando el lenguaje de programación C, que es bastante más rápido que Python.

Ahora definamos la librería….

NumPy es una librería de Python especializada en el cálculo numérico y el análisis de datos, especialmente para un gran volumen de datos.

Incorpora una nueva clase de objetos llamados **arrays** que permite representar colecciones de datos de un mismo tipo en varias dimensiones, y funciones muy eficientes para su manipulación

La clase de objetos array

Un array es una estructura de datos de un mismo tipo organizada en forma de tabla o cuadrícula de distintas dimensiones.

Las dimensiones de un array también se conocen como ejes.

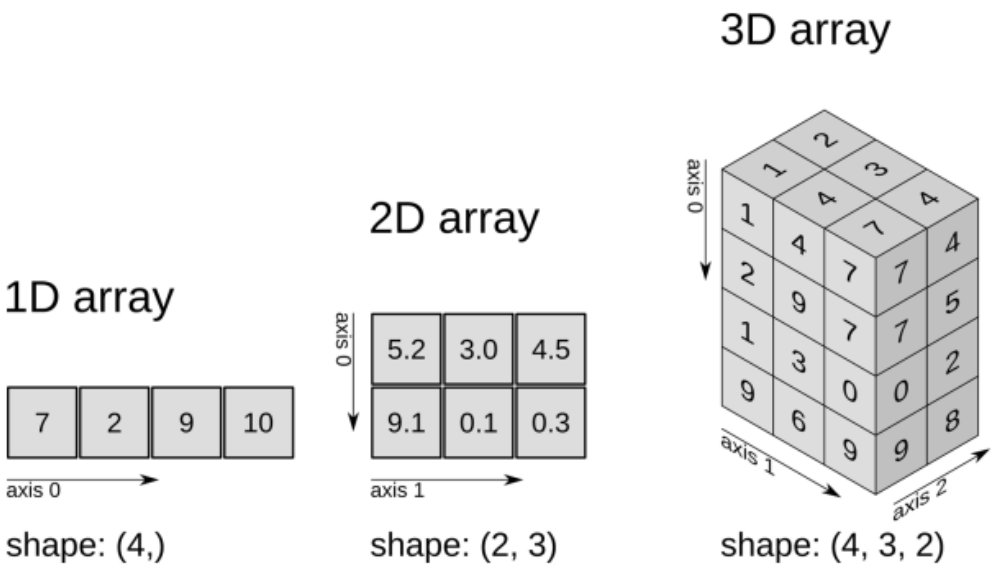

imagen:fuente propia

Creación de arrays

Para crear un array se utiliza la siguiente función de NumPy. primero debemos importarla para esto utilizamos la siguiente línea de código:

import numpy as np

"**np**" es la convención que se utiliza para emplear NumPy dentro del código.

np.array(lista)

Crea un array a partir de la lista o tupla lista y devuelve una referencia a él. El número de dimensiones del array dependerá de las listas o tuplas anidadas en **lista**:

- Para una lista de valores se crea un array de una dimensión, también conocido como vector.
- Para una lista de listas de valores se crea un array de dos dimensiones, también conocido como matriz.
- Para una lista de listas de listas de valores se crea un array de tres dimensiones, también conocido como cubo.
- Y así sucesivamente. No hay límite en el número de dimensiones del array más allá de la memoria disponible en el sistema.

Los elementos de la lista o tupla deben ser del mismo tipo.

```
# Array de una dimensión

import numpy as np

a1 = np.array([1, 2, 3])
print(a1)

# [1 2 3]

# Array de dos dimensiones

a2 = np.array([[1, 2, 3], [4, 5, 6]])
print(a2)

# [[1 2 3]

   [4 5 6]]

# Array de tres dimensiones

a3 = np.array([[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [1, 2, 3]])
print(a3)

# [[1 2 3]

   [4 5 6]]
```

```
  [[7 8 9]
   [1 2 3]]
```

Otras funciones útiles que permiten generar arrays son:

np.empty(dimensiones)

Crea y devuelve una referencia a un array vacío con las dimensiones especificadas en la tupla dimensiones.

np.zeros(dimensiones)

Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla dimensiones cuyos elementos son todos ceros.

```
print(np.zeros(3,2))
# [[0. 0. 0.]
   [0. 0. 0.]]
```

np.ones(dimensiones)

Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla dimensiones cuyos elementos son todos unos.

np.full(dimensiones, valor)

Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla dimensiones cuyos elementos son todos valor.

np.identity(n)

Crea y devuelve una referencia a la matriz identidad de dimensión n.

```
print(np.idendity(3))

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

np.arange(inicio, fin, salto)

Crea y devuelve una referencia a un array de una dimensión cuyos elementos son la secuencia desde inicio hasta fin tomando valores cada salto.

```
print(np.arange(1, 10, 2))

# [1, 3, 5, 7, 9]
```

np.linspace(inicio, fin, n)

Crea y devuelve una referencia a un array de una dimensión cuyos elementos son la secuencia de n valores equidistantes desde inicio hasta fin.

```
print(np.linspace(0, 10, 5))

# [0.  2.5 5.  7.5 10.]
```

np.random.random(dimensiones)

Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla dimensiones cuyos elementos son aleatorios.

Atributos de los arrays

Existen varios atributos y funciones que describen las características de un array.

a.ndi : Devuelve el número de dimensiones del array a.

a.shape : Devuelve una tupla con las dimensiones del array a.

a.size : Devuelve el número de elementos del array a.

a.dtype: Devuelve el tipo de datos de los elementos del array a.

Selección de elementos de un arreglo.

Cómo seleccionar elementos en un Array de Numpy. Tanto en vectores unidimensionales como de más dimensiones. Posiblemente la selección de elementos es una de las tareas que se realiza con mayor frecuencia con los objetos de Numpy.

Para utilizar como ejemplo durante el resto de la entrada se creará un Array de Numpy con la función **arrage**() como se muestra a continuación.

```
import numpy as np

array = np.arange(1, 29, 3)
array

# array([ 1,  4,  7, 10, 13, 16, 19, 22, 25, 28])
```

Seleccionar un único elemento en un Array de Numpy

Para seleccionar un único elemento en un Array de Numpy se indica su posición entre corchetes después del nombre del objeto. Recordando que en Python la posición de los elementos empieza a contar el 0. Así, para obtener el segundo elemento del vector creado anteriormente se puede utilizar la siguiente línea de código.

```
array[1]

# 4
```

En donde se obtiene el valor 4 ya que se ha indicado el segundo elemento del vector, recordando otra vez que en Python el primer elemento es 0 no 1.

Seleccionar una parte de un Array de Numpy

Los corchetes no solamente permiten seleccionar un único elemento, sino que se pueden seleccionar una parte del objeto. Lo que se puede hacer mediante el uso del operador dos puntos. Indicando entre corchetes la posición del primer elemento, dos puntos y la posición posterior a la última que se desea seleccionar. Recordando que el operador dos puntos función de forma análoga a arange, generando un vector desde la posición inicial hasta al anterior a la final. Esto se puede ver en un ejemplo en el que se selecciona los valores del segundo al quinto.

```
array[2:6]

# array([7, 10, 13, 16])
```

Con lo que se obtiene el resultado esperado.

Una opción interesante que tiene el operador dos puntos es la posibilidad de omitir tanto el punto de inicio como el final. Así el vector se seleccionará desde el inicio hasta el final o desde el punto de inicio hasta el final. Por ejemplo, para ver los puntos desde el inicio hasta el quinto elemento se puede escribir

```
array[:5]

# array([1, 4, 7 , 10, 13])
```

Por otro lado, para seleccionar desde el sexto elemento hasta el final se puede utilizar la siguiente línea

```
array[:5]
```

```
# array([16, 19, 22, 25, 28])
```

Es posible omitir tanto el punto de inicio como el final, con lo que se creará una copia del vector original.

Seleccionar los elementos en una matriz.

Los elementos de una matriz se seleccionan prácticamente igual que en el caso de los vectores. Lo único que hay que tener en cuenta es que ahora el objeto tiene más dimensiones, por lo que es necesario suministrar esos dos valores separados por comas. En primer lugar, es necesario indicar la coordenada de la fila y el segundo el de la columna. Así se puede crear una matriz y seleccionar segundo elemento de la primera fila con el siguiente comando

```
array = np.arange(1,10).reshape(3,3)
array

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

Al igual que el caso unidimensional se puede utilizar el operador dos puntos para seleccionar un grupo de una fila o columna. O está en su conjunto si no se indica punto de inicio ni de fin. Por ejemplo, la segunda fila de la matriz se puede obtener con

```
array[1, :]

# array([4, 5, 6])
```

Otros Ejemplos:

```
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a[1,0]) # Acceso al elemento de la fila 1 columna 0

# 4

print(a[1][0]) # Otra forma de acceder al mismo elemento

# 4

print(a[:, 0:2])

# [[1 2]

   [4 5]]
```

Selección Condicional de elementos en un arreglo

Una característica muy útil de los arrays es que es muy fácil obtener otro array con los elementos que cumplen una condición.

a[condicion] : Devuelve una lista con los elementos del array a que cumplen la condición.

```
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a[(a % 2 == 0)])

# [2 4 6]

print(a[(a % 2 == 0)] & (a > 2))
# [2 4]
```

Operaciones matemáticas con arrays

Existen dos formas de realizar operaciones matemáticas con arrays: a nivel de elemento y a nivel de array.

Las operaciones a nivel de elemento operan los elementos que ocupan la misma posición en dos arrays. Se necesitan, por tanto, dos arrays con las mismas dimensiones y el resultado es una array de la misma dimensión.

Los operadores matemáticos +, -, *, /, %, ** Se utilizan para la realizar suma, resta, producto, cociente, resto y potencia a nivel de elemento.

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

# [[2 3 4]

   [6 7 8]]

print(a / b)

# [[1. 2.  3.]

   [2. 2.5 3.]]

print(a ** 2)

# [[1  4   9]

   [16 25 36]]
```

Operaciones matemáticas a nivel de arrays

Para realizar el producto matricial se utiliza el método

a.dot(b) : Devuelve el array resultado del producto matricial de los arrays a y b siempre y cuando sus dimensiones sean compatibles.

Y para trasponer una matriz se utiliza el método

a.T : Devuelve el array resultado de trasponer el array a.

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

# [[14 14]

   [32 32]]

print(a.T)

# [[1 4]

   [2 5]

   [3 6]]
```

Redimensionar arreglos

NumPy tiene dos funciones (y también métodos) para cambiar las formas de los arrays - **reshape** y **resize**

np.reshape()

Esta función devuelve un nuevo array con los datos del array cedido como primer argumento y el nuevo tamaño indicado:

```
import numpy as np

arrayA = np.arange(8)

# arrayA = array([0, 1, 2, 3, 4, 5, 6, 7])

np.reshape(arrayA,(2,4))

# array([[0, 1, 2, 3],

         [4, 5, 6, 7]])
```

Convierte un vector de 8 elementos a el array de la forma de **(4, 2)**. Puede ser ejecutada con éxito porque la cantidad de elementos antes y después de reshape es idéntica. Aumenta el **ValueError** si las cantidades son diferentes.

```
arrayA = array([0, 1, 2, 3, 4, 5, 6, 7])
np.reshape(arrayA,(3,4))

# ValueError: cannot reshape array of size 8 into shape (3,4)
```

La primera fila contiene los primeros 4 datos del **arrayA** y la segunda fila contiene los últimos 4. Rellena los datos en el orden de la fila en esta conversión de remodelación.

Necesitas cambiar el parámetro order si quieres que el **orden** de llenado de los datos sea de columna.

```
np.reshape(arrayA,(2,4), order = "F")

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

El valor por defecto de **orden** es **C**, qué significa leer o escribir datos en un orden de índice similar a **C**, o en palabras simples, en el orden de la fila. **F** significa leer o escribir datos en un orden de índice similar al Fortan, o digamos, en el orden de la **column**.

nsarray.reshape()

Además de la función **reshape**, NumPy tiene también el método **reshape** en el objeto **ndarray**. El método tiene los mismos parámetros que la función pero sin el array como parámetro.

```
arrayB = arrayA.reshape((2,4))

print(arrayA)

# [0 1 2 3 4 5 6 7]

print(arrayB)

# [[0 1 2 3]

   [4 5 6 7]]
```

np.resize()

Es un poco similar a **reshape** en el sentido de conversión de formas. Pero tiene algunas diferencias significativas.

1. No tiene el parámetro order. El orden de resize es el mismo que order='C' en reshape.
2. Si el número de elementos del array de destino no es el mismo que el del array original, forzará a redimensionar pero no provocará errores.

Centrémonos en la segunda diferencia

```
arrayA = np.arange(8)
arrayB = np.resize(arrayA, (2,4))
```

El resultado es el mismo que en **reshpae** si los números de elementos son los mismos.

```
arrayD = np.resize(arrayA, (4,4))
arrayD

# array([[0, 1, 2, 3],

         [4, 5, 6, 7],

         [0, 1, 2, 3],

         [4, 5, 6, 7]])
```

Si el nuevo array tiene más filas, repetirá los datos del array original pero no aumentará el error.

```
arrayE = np.resize(arrayA, (2,2))
arrayE

# array([[0, 1],

         [2, 3]])

np.resize(arrayA, (1,4))

# array([[0, 1, 2, 3]])
```

Si el número de elementos en la nueva matriz es menor, se obtiene el número de elementos que necesita para rellenar la nueva matriz en el orden de las filas.

Referencias

[1] Librería NumPy

https://aprendeia.com/introduccion-a-numpy-python-1/

https://imalexissaez.github.io/2018/08/18/breve-introduccion-a-la-libreria-numpy/

https://numpy.org/doc/stable/reference/generated/numpy.identity.html

[2] Operaciones entre arreglos

https://claudiovz.github.io/scipy-lecture-notes-ES/intro/numpy/operations.html