prog_datasci_4_numpy

November 1, 2019

1 Programación para *Data Science*

1.1 Unidad 4: Librerías científicas en Python - NumPy

1.1.1 Instrucciones de uso

A continuación se presentarán explicaciones y ejemplos de uso de la librería NumPy. Recordad que podéis ir ejecutando los ejemplos para obtener sus resultados.

1.1.2 Primeros pasos

Vamos a importar la librería:

[1]:
```
# En la siguiente línea, importamos NumPy y le damos un nombre más corto
# para que nos sea más cómodo hacer las llamadas.
import numpy as np
```

En NumPy, el objeto básico se trata de una lista multidimensional de números (normalmente) del mismo tipo.

[2]:
```
# Ejemplo básico, un punto en el espacio:
p = np.array([1, 2, 3])
```

En NumPy, a las dimensiones se les conoce con el nombre de ejes (*axes*) y al número de ejes, rango (*rank*). *array* es un alias para referirse al tipo de objeto *numpy.ndarray*.

Algunas propiedades importantes de los *arrays* son las siguientes: * **ndarray.ndim**: el número de ejes del objeto *array* (matriz) * **ndarray.shape**: una tupla de números enteros indicando la longitud de las dimensiones de la matriz * **ndarray.size**: el número total de elementos de la matriz

[3]:
```
# Vamos a crear una matriz bidimensional 3x2 (tres filas, dos columnas).
a = np.arange(3*2) # Creamos un array unidimensional de 6 elementos.
print('Array unidimensional:')
print(a)
a = a.reshape(3,2) # Le damos "forma" de matriz 3x2.
print('Matriz 3x2:')
print(a)
```

```
Array unidimensional:
[0 1 2 3 4 5]
Matriz 3x2:
[[0 1]
```

```
[2 3]
[4 5]]
```

[4]:
```
# La dimensión es 2.
a.ndim
```

[4]:
```
2
```

[5]:
```
# Longitud de las dimensiones
a.shape
```

[5]:
```
(3, 2)
```

[6]:
```
# El número total de elementos
a.size
```

[6]:
```
6
```

A la hora de crear arrays, tenemos diferentes opciones:

[7]:
```
# Creamos un array (vector) de diez elementos:
z = np.zeros(10)
print(z)
```

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

[8]:
```
# Podemos cambiar cualquiera de los valores de este vector accediendo a su␣
 ↪posición:
z[4] = 5.0
z[-1] = 0.1
print(z)
```

```
[0.  0.  0.  0.  5.  0.  0.  0.  0.  0.1]
```

[9]:
```
# La función arange nos permite definir diferentes opciones, como el punto de␣
 ↪inicio y el de fin:
a = np.arange(10,20)
print(a)
```

```
[10 11 12 13 14 15 16 17 18 19]
```

[10]:
```
# El último argumento nos permite utilizar un paso de 2:
a = np.arange(10,20,2)
print(a)
```

```
[10 12 14 16 18]
```

[11]:
```
# Podemos crear arrays desde listas de Python de varias dimensiones:
a = np.array([[1, 2, 3], [4, 5, 6]])
```

```
print(a)
```

```
[[1 2 3]
 [4 5 6]]
```

1.1.3 Operaciones con matrices

NumPy implementa todas las operaciones habituales con matrices.

[12]:
```
A = np.array([[1,0], [0,1]])
B = np.array([[1,2], [3,4]])
```

[13]:
```
# Suma de matrices
print(A+B)
```

```
[[2 2]
 [3 5]]
```

[14]:
```
# Resta de matrices
print(A-B)
```

```
[[ 0 -2]
 [-3 -3]]
```

[15]:
```
# Multiplicación elemento por elemento
print(A*B)
```

```
[[1 0]
 [0 4]]
```

[16]:
```
# Multiplicación de matrices
print(A.dot(B))
```

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

[17]:
```
# Potencia
print(B**2)
```

```
[[ 1  4]
 [ 9 16]]
```

1.1.4 *Slicing* e iteración

Los *arrays* en NumPy soportan la técnica de *slicing* de Python. Podemos usar esta técnica para recuperar valores de un array:

[18]:
```
# Definimos un array de 0 a 9.
a = np.arange(10)
print(a)

# Obtenemos los 5-2 elementos desde la posición 3 del array (los índices␣
 ↪empiezan en 0).
print(a[2:5])
```

```
[0 1 2 3 4 5 6 7 8 9]
[2 3 4]
```

[19]:
```
# Todos los elementos a partir de la tercera posición
a[2:]
```

[19]:
```
array([2, 3, 4, 5, 6, 7, 8, 9])
```

[20]:
```
# Podemos iterar por cada elemento del array:
for i in a:
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

[21]:
```
# Definamos ahora un array multidimensional.
A = np.arange(18).reshape(6,3)
print(A)
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]
 [15 16 17]]
```

[22]:
```
# Accedemos a una posición concreta del array multidimensional

# Primera alternativa:
print(A[3][2])
# Segunda alternativa
```

```
print(A[3, 2])
```

```
11
11
```

[23]:
```
# Accedim a una fila completa (en este caso, la fila 2)
print(A[2])
```

```
[6 7 8]
```

[24]:
```
# Ahora a una columna completa (en este cas0, la columna 0)
print(A[:, 0])
```

```
[ 0  3  6  9 12 15]
```

[25]:
```
# Obtenemos todas las filas hasta la quinta fila
print(A[:5])
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]
```

[26]:
```
# Podemos iterar a través de los elementos del array multidimensional
for i in range(A.shape[0]):
    for j in range(A.shape[1]):
        print("Elemento: {}".format(A[i, j]))
```

```
Elemento: 0
Elemento: 1
Elemento: 2
Elemento: 3
Elemento: 4
Elemento: 5
Elemento: 6
Elemento: 7
Elemento: 8
Elemento: 9
Elemento: 10
Elemento: 11
Elemento: 12
Elemento: 13
Elemento: 14
Elemento: 15
Elemento: 16
Elemento: 17
```

También podemos usar *slicing* en asignaciones:

[27]:
```
# Modificamos el valor de la posición 3, 2
A[3, 2] = 42
print(A)
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 42]
 [12 13 14]
 [15 16 17]]
```

[28]:
```
# Modificamos el valor de una fila completa
A[2] = [101, 101, 101]
print(A)
```

```
[[  0   1   2]
 [  3   4   5]
 [101 101 101]
 [  9  10  42]
 [ 12  13  14]
 [ 15  16  17]]
```

[23]:
```
# Igualamos todos los elementos a 0 (el operador ... añade todos los :␣
 ↪necesarios).
A[...] = 0
print(A)
```

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

1.1.5 Ejemplo: El juego de la vida de Conway

El juego de la vida es un ejemplo clásico de autómata celular creado en 1970 por el famoso matemático John H. Conway. En el problema clásico, se representan en una matriz bidimensional células que vivirán o morirán dependiendo del número de vecinos en un determinado paso de la simulación. Cada célula tiene 8 vecinos (las casillas adyacentes en un tablero bidimensional) y puede estar viva (1) o muerta (0). Las reglas clásicas entre transición vida/muerte son las siguientes:

- Una célula muerta con exactamente 3 células vecinas vivas al turno siguiente estará viva.
- Una célula viva con 2 o 3 células vecinas vivas sigue viva. Si no es el caso, muere o permanece muerta (soledad en caso de un número menor a 2, superpoblación si es mayor a 3).

Este problema (o juego) tiene muchas variantes dependiendo de las condiciones iniciales, si el tablero (o mundo) tiene bordes o no, o si bien las reglas de vida o muerte son alteradas.

A continuación tenéis un ejemplo del juego clásico implementado en NumPy:

[24]:
```
# Autor: Nicolas Rougier
# Fuente: http://www.labri.fr/perso/nrougier/teaching/numpy.100/

SIZE = 10
STEPS = 10

def iterate(Z):
    # Count neighbours
    N = (Z[0:-2,0:-2] + Z[0:-2,1:-1] + Z[0:-2,2:] +
         Z[1:-1,0:-2]                + Z[1:-1,2:] +
         Z[2:  ,0:-2] + Z[2:  ,1:-1] + Z[2:  ,2:])

    # Apply rules
    birth = (N==3) & (Z[1:-1,1:-1]==0)
    survive = ((N==2) | (N==3)) & (Z[1:-1,1:-1]==1)
    Z[...] = 0
    Z[1:-1,1:-1][birth | survive] = 1
    return Z

# Creamos un tablero con células vivas o muertas de forma aleatoria.
Z = np.random.randint(0,2,(SIZE, SIZE))
# Simulamos durante los pasos indicados.
for i in range(STEPS):
    Z = iterate(Z)
# Mostramos el tablero en el paso final.
print(Z)
```

```
[[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]
 [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]]
```