prog_datasci_4_numpy

September 24, 2022

1 Fundamentos de Programación

1.1 Unidad 4: Librerías científicas en Python - NumPy

1.1.1 Instrucciones de uso

Este documento es un *notebook* interactivo que intercala explicaciones más bien teóricas de conceptos de programación con fragmentos de código ejecutables. Para aprovechar las ventajas que aporta este formato, se recomienda, en primer lugar, leer las explicaciones y el código que os proporcionamos. De esta manera tendréis un primer contacto con los conceptos que exponemos. Ahora bien, **¡la lectura es sólo el principio!** Una vez hayáis leído el contenido, no olvidéis ejecutar el código proporcionado y modificarlo para crear variantes que os permitan comprobar que habéis entendido su funcionalidad y explorar los detalles de implementación. Por último, se recomienda también consultar la documentación enlazada para explorar con más profundidad las funcionalidades de los módulos presentados.

Para guardar posibles modificaciones que hagáis sobre este notebook, os aconsejamos que montéis la unidad de Drive en Google Colaboratory (colab). Tenéis que ejecutar las instrucciones siguientes:

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

[ ]:
```
%cd /content/drive/MyDrive/Colab_Notebooks/prog_datasci_4
```

1.1.2 Introducción

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

A continuación se incluye la tabla de contenidos, que podéis utilizar para navegar por el documento:

1 Introducción

NumPy nació con el nombre de Numeric y, poco a poco, fue centralizando muchos otros desarrollos de otros autores (como NumArray) para acabar siendo la librería que hoy día conocemos como NumPy. En la actualidad NumPy está integrada en la SciPy stack aunque continúa siendo un paquete independiente de SciPy (más tarde hablaremos de este paquete).

NumPy es una librería de cálculo matricial en origen, aunque hoy en día implementa muchas otras funcionalidades interesantes como son herramientas para integrar código C/C++ y Fortran, funciones de álgebra lineal, transformaciones de Fourier, etc.

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]]
```

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]]
```

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]]
```

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]]
```

6 Ejercicios y preguntas teóricas

La parte evaluable de esta unidad consiste en la entrega de un fichero Notebook con extensión «.ipynb» que contendrá los diferentes ejercicios y las preguntas teóricas que hay que contestar. Encontraréis el archivo (`prog_datasci_4_scilib_entrega.ipynb`) con las actividades en la misma carpeta que este notebook que estáis leyendo. **Hay un solo archivo de actividades para toda la unidad, que cubre todas las librerías que se trabajan.**

6.1. Instrucciones importantes Es muy importante que a la hora de entregar el fichero Notebook con vuestras actividades os aseguréis de que:

1. Vuestras soluciones sean originales. Esperamos no detectar copia directa entre estudiantes.
2. Todo el código esté correctamente documentado. El código sin documentar equivaldrá a un 0.
3. El fichero comprimido que entregáis es correcto (contiene las actividades de la PEC que tenéis

que entregar).

Para hacer la entrega, tenéis que ir a la carpeta del drive `Colab Notebooks`, clicando con el botón derecho en la PEC en cuestión y haciendo `Download`. De este modo, os bajaréis la carpeta de la PEC comprimida en `zip`. Este es el archivo que tenéis que subir al campus virtual de la asignatura.

Una vez descargada la solución, podéis ejecutar el servidor como ya explicamos (mediante el script `start_uoc.sh`) y acceder a su contenido.

2 Autores

- Autor original Brian Jiménez Garcia, 2016.
- Actualizado por Cristina Pérez Solà, 2017 y 2019.


```
<div style="width:0%;"> </div>
    <img src="https://mirrors.creativecommons.org/presskit/buttons/88x31/png/by-sa.png">
```