Numpy arrays

Tipos de datos

Los **argumentos** que se deben indicar para crear el array son dos: los **elementos que contendrá**, que se ingresan en forma de lista, y el **tipo de datos** que tendrá la lista. En los ejemplos anteriores no especificamos el tipo de datos, ya que éste se asigna automáticamente al crear la lista. No obstante, es recomendable indicarlo siempre ya que cada tipo de datos ocupa un espacio en memoria distinto.

Los tipos de datos que podemos utilizar dentro de un array son los siguientes: * **Números enteros** de 8, 16, 32 y 64 bits, con signo y sin signo. * **Números complejos** de 64 y 128 bits. * **Números de punto flotante** de 16, 32, 64 y 128 bits * **Booleanos** ('True' o 'False'), **strings**, **bytes...**

En el caso del array 'a' que creamos anteriormente, el formato que se asigna automáticamente es un entero de 64 bits con signo (int64). Nos resulta conveniente utilizar un formato entero sin signo y de 8 bits, de acuerdo a los datos que contiene. El formato se especifica de la siguiente forma:

[1]:
```
import numpy as np

b = np.array([1,2,5,9], dtype = np.uint8)

# la consola nos muestra el array
b
```

[1]:
```
array([1, 2, 5, 9], dtype=uint8)
```

b.dtype

Longitud y dimensiones de un array

Los ejemplos mencionados hasta ahora han sido de arrays de dimensiones $1xN$. Podemos crear **arrays de varias dimensiones** de la siguiente manera:

[2]:
```
# array de dos dimensiones y longitud igual a 3
b = np.array([[1,2,3],[3,2,1]], dtype = np.uint8)

# array de tres dimensiones y longitud igual a 2
c = np.array([[1,-1],[0,1],[-5,2]])
```

La **longitud** representa la cantidad de elementos que posee el array. Su valor, y el de las dimensiones de un array, se pueden obtener de la siguiente manera:

[3]:
```
# devuelve la forma de un array, de la siguiente forma (n° de filas, n° de␣
↪columnas)
(dim1, dim2) = b.shape

#longitud del array
longitud = b.size

print('numero de filas: ', dim1, '\nnumero de columnas: ', dim2)
print('longitud del array: ', longitud)
```

```
numero de filas:  2
numero de columnas:  3
longitud del array:  6
```

Indizado en arrays

Los **índices** se utilizan para indicar la posición de cada elemento de un array. De esta manera, podemos acceder al valor de una posición específica o extraer el valor de un grupo de elementos. Cada elemento posee una posición $[i, j]$, donde i representa el número de fila y j representa el número de columna en que se ubica.

[4]:
```
# devuelve el elemento de la posición [1,2] del array 'a', igual a 2.
print(b[1,2])

# devuelve el elemento de la posición [2,0] del array 'b', igual a -5.
print(c[2,0])
```

```
1
-5
```

Es importante tener en cuenta que los índices que se asignan se inician desde 0, por lo que si tengo 4 elementos, sus índices irán del rango de 0 a 3. Podemos **desplazarnos** dentro de una array, y **seleccionar** un rango de elementos utilizando la notación *[i: j: k]*, donde 'i' representa el índice a partir del cual nos desplazamos, 'j' es el índice en el cual nos detenemos y 'k' es el paso con el cual nos desplazamos a los elementos subsiguientes.

[5]:
```
s = [0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5]
sec = np.array(s, dtype = np.float16)

# desplazamiento a partir del 1° elemento, hasta el 8° (no inclusive),
# en pasos de cada 2 elementos
print (sec[0:8:2])

# desplazamiento entre las mismas posiciones en pasos de a 1
print (sec[:8])

# desplazamiento desde la posición n-5 hasta la posición 9, siendo n la cantidad␣
 ↪de elementos.
print (sec[-5:9])

# desplazamiento desde la posición 7 hasta la posición n-8, en pasos de 1 hacia␣
 ↪atrás
print (sec[7:-8:-1])

# desplazamiento desde la posición 4 hasta el final del array
print (sec[3:])

# desplazamiento desde la posición 4 hasta el final del array en pasos de a tres
print (sec[3::3])
```

```
# desplazamiento desde el comienzo hasta la posición 6 en pasos de a 2
print (sec[:6:2])
```

```
[0. 1. 2. 3.]
[0.  0.5 1.  1.5 2.  2.5 3.  3.5]
[2.5 3.  3.5 4. ]
[3.5 3.  2.5 2.  1.5]
[1.5 2.  2.5 3.  3.5 4.  4.5]
[1.5 3.  4.5]
[0. 1. 2.]
```

Como se puede ver en el ejemplo, tenemos varias opciones para desplazarnos de distintas maneras. Es posible ir hacia adelante, atrás e indicar las posiciones inicial y final con respecto a la longitud del array. Notemos que el elemento j, que indica el fin del desplazamiento, no se incluye en el resultado. Es decir, el resultado nos arroja los **elementos desde** i **hasta** $j-1$.

Para el caso de **matrices**, el desplazamiento es similar, contando con algunas opciones adicionales. Veamos algunos casos.

[6]:
```
# definimos la misma secuencia, pero en forma de matriz de 3x3
fibo_mtx = np.array([[0,1,1],[2,3,5],[8,13,21]], dtype = np.uint8)

# el desplazamiento es ahora entre dimensiones, para ver las dos primeras (filas␣
 ↪0 y 1):
print(fibo_mtx[0:2], "\n")

# veo elementos de la submatriz de 2x2 (primeras dos filas y dos columnas):
print(fibo_mtx[0:2,0:2], "\n")

# desplazamiento entre los dos primeros elementos de la 3° fila
print(fibo_mtx[2,0:2], "\n")

# desplazamiento entre filas:
print(fibo_mtx[0:2,0::], "\n")

# desplazamiento entre los elementos de la 2° columna
print(fibo_mtx[...,1])
```

```
[[0 1 1]
 [2 3 5]]

[[0 1]
 [2 3]]

[ 8 13]

[[0 1 1]
```

```
 [2 3 5]]

[ 1  3 13]
```

La notación *[i , j, k]* sirve para desplazarnos entre filas de una matriz. Por otra parte, podemos acceder a todos los elementos de una columna o fila con la notación *[..., x]* o *[x, ...]* respectivamente, siendo *x* el índice de la columna o fila en la que nos desplazamos.

> **Nota:** Numpy también cuenta con el objeto matrix, que es una subclase del objeto array. La ventaja de usar este objeto es que las operaciones matemáticas son matriciales por defecto, y en un array se hacen elemento a elemento (vamos a este tema en la siguiente sección). A fines prácticos, el array también se puede implementar como matriz, teniendo en consideración lo mencionado anteriormente.

0.1 Broadcasting

Sólo es posible operar entre arrays bajo una de las siguientes condiciones:

- Ambos arrays tienen *m* x *n* dimensiones
- Un array tiene 1 x *n* dimensiones y el otro tiene *m* x *n*

En el segundo caso, la operación entre arrays se realiza igualmente, aunque no coincidan en una de sus dimensiones. Mediante el broadcasting, el array más pequeño se replica *m* veces para poder operar con el más grande. El término alude a cómo Numpy realiza las operaciones con arrays. En español, se podría traducir como 'expansión' ya que describe lo que sucede al operar entre arrays de distintos tamaños, donde **el array más pequeño se expande a lo largo del array más grande** para poder realizar la operación. Esto se hace automáticamente y puede ser aprovechado para realizar operaciones en pocos pasos, sin necesidad de aplicar iteraciones.

[7]:
```
# Veamos en detalle cómo funciona el broadcasting

# partimos de estos dos vectores
t = np.array([1,2,3,4], dtype = np.int8)
u = np.array([0,5,10], dtype = np.int8)

t.shape, u.shape
t, u
```

[7]:
```
(array([1, 2, 3, 4], dtype=int8), array([ 0,  5, 10], dtype=int8))
```

[8]:
```
t + u
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-8-449f4ade2cf0> in <module>
----> 1 t + u

ValueError: operands could not be broadcast together with shapes (4,) (3,)
```

En el ejemplo, las longitudes de a y b son distintas. A pesar de que el comando `shape` nos muestra sólo una de las longitudes, podemos trabajar considerando que son de la forma (4,1) y (3,1). Vemos que la primer longitud de ambos es distinta (4 y 3), mientras que la segunda longitud es igual (1 y 1). Ninguna operación puede hacerse bajo estas condiciones. Sin embargo, podemos manipular la forma de ellos para que sea posible operar. Si aplicamos `reshape` a 'u' para que sea de la forma (1,3), sus longitudes serán compatibles. Por un lado tendremos un array de la forma 4 x 1 y por el otro uno de la forma 1 x 3. Todavía son distintas pero sucede que es posible operar por haber adecuado la forma de uno de los vectores. Ahora vemos lo siguiente: * 'u' tiene 1 columna, 't' tiene 4 columnas --> 'u' se expande para tener 4 columnas * 'u' tiene 3 filas, 't' tiene 1 fila --> 'v' se expande para tener tres filas

[9]:
```
# cambiamos la forma de 'u' para poder operar
u = u.reshape(3,1)
print(u)
```

```
[[ 0]
 [ 5]
 [10]]
```

[10]:
```
t + u
```

[10]:
```
array([[ 1,  2,  3,  4],
       [ 6,  7,  8,  9],
       [11, 12, 13, 14]], dtype=int8)
```

Lo anterior es equivalente a realizar la siguiente operación entre arrays de iguales dimensiones:

[11]:
```
t2 = np.array([[1,2,3,4],[1,2,3,4],[1,2,3,4]], dtype=np.int8)
u2 = np.array([[0,0,0,0],[5,5,5,5],[10,10,10,10]], dtype=np.int8)
```

[12]:
```
t2 + u2
```

[12]:
```
array([[ 1,  2,  3,  4],
       [ 6,  7,  8,  9],
       [11, 12, 13, 14]], dtype=int8)
```

Otras formas de crear arrays

Numpy provee funciones para crear ciertos tipos de arrays de común uso. Veamos algunos casos:

[13]:
```
# creamos un array de unos
array_unos = np.ones((2,6), dtype = np.uint8)

# creamos un array de ceros
array_ceros = np.zeros((3,3), dtype = np.uint8)

# creamos una secuencia creciente de números
sec_nros = np.arange(10, dtype = np.uint8)
```

```
# otra secuencia de números
otra_sec_nros = np.linspace(0,5,5, dtype=np.float16)

print(array_unos, "\n")
print(array_ceros, "\n")
print(sec_nros, "\n")
print(otra_sec_nros, "\n")
```

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

[[0 0 0]
 [0 0 0]
 [0 0 0]]

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

[0.   1.25 2.5  3.75 5.  ]
```

Veamos algunas formas de crear secuencias aleatorias:

[14]:
```
# randint = enteros aleatorios - se especifica el rango de valores

rand_sec = np.random.randint(-100, 100, 8, dtype=np.int8)
print(rand_sec)
```

```
[ 88 -37  33  -7 -57 -59  79 -94]
```

[15]:
```
# rand = array con valores aleatorios entre 0 y 1 - se especifican las␣
 ↪dimensiones y
# longitud del array

rand_sec = np.random.rand(2,3)
print(rand_sec)
```

```
[[0.30969422 0.70287528 0.60202102]
 [0.81748186 0.53348233 0.85071785]]
```

[16]:
```
# ranf = floats aleatorios - se especifica la longitud

rand_sec = np.random.ranf(5)
print(rand_sec)
```

```
[0.95112166 0.17795099 0.36370031 0.62058227 0.56775821]
```

Aplicando funciones matematicas

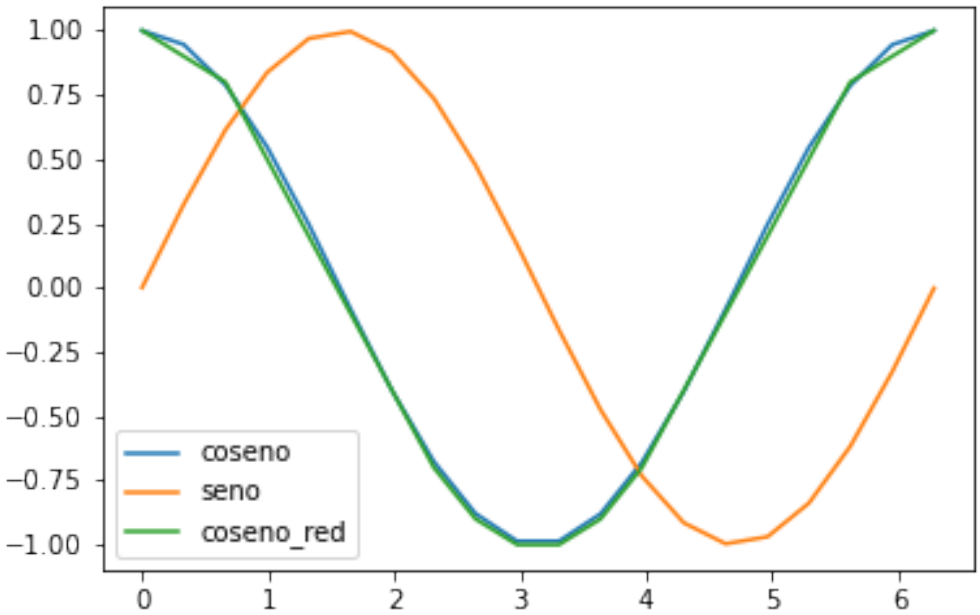
Numpy dispone de varias funciones matemáticas para aplicar sobre un simple escalar o sobre un array. La lista completa de funciones matemáticas que provee Numpy puede verse haciendo clic acá.

Veamos un ejemplo:

[17]:
```
# multiplico una array por el escalar pi.
x = np.linspace(0., 2., 20, dtype = np.float16)*np.pi

# funciones seno y coseno
f_seno = np.sin(x)
f_coseno = np.cos(x)

# redondeo los valores del coseno
f_coseno_red = np.around(f_coseno, 1)
```

[18]:
```
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(x, f_coseno, label='coseno')
plt.plot(x, f_seno, label='seno')
plt.plot(x, f_coseno_red, label='coseno_red')
plt.legend()
```

[18]:
```
<matplotlib.legend.Legend at 0x7fccbff87910>
```



Entonces, podemos crear arrays para representar funciones matemáticas conocidas. Primero creamos un array para definir una serie de puntos en el dominio de la función y luego aplicamos la función sobre estos puntos.

Una de las funciones que utilizamos fue *linspace*. Es una función práctica para crear una representación de una función o un conjunto de datos. En este caso, con x tenemos representados los puntos en el dominio del tiempo, y aplicando la función sobre esos puntos creamos una array que representa a la misma.

Funciones definidas por el usuario

Una función, es la forma de **agrupar expresiones y sentencias** que realicen determinadas acciones, pero que éstas, solo **se ejecuten cuando son llamadas**. Es decir, que al colocar un algoritmo dentro de una función y se corre el archivo, el algoritmo no será ejecutado si no se ha hecho una referencia a la función que lo contiene. En definitiva lo más importante para programar, y no solo en Python, es saber organizar el código en piezas más pequeñas que hagan tareas independientes y combinarlas entre sí. Las funciones son el primer nivel de organización del código: reciben unas entradas, las procesan y devuelven unas salidas.

Las funciones tienen un gran potencial, para conocer más detalles.

Definicion de funciones

Lo vemos con un ejemplo concreto:

[19]:
```
# -*- coding: utf-8 -*-

def fib(n):
    """ Escribe la serie de Fibonacci hasta n. """

    a, b = 0, 1  # asignacion multiple
    while a < n:
        print (a, end = ' ') # es para imprimirlos seguidos y no uno debajo del␣
↪otro.
```

```
        a, b = b, a + b
```

[20]:
```
?fib(n)
```

```
Object `fib(n)` not found.
```

[21]:
```
# Invocamos la función definida.
fib(2000)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

La palabra reservada `def` se usa para **definir funciones**. Debe seguirle el nombre de la función y la **lista de argumentos entre paréntesis**. Las sentencias que forman el cuerpo de la función empiezan en la línea siguiente, y deben estar con sangría. **La primer sentencia** del cuerpo de la función puede ser opcionalmente una cadena de texto literal (utilizar """, en lugar de #); esta es la cadena de texto de documentación de la función, o **docstring**. Es una buena práctica, no solo documentar las funciones, sino hacerlo con un estilo único y estandarizado. Una referencia respaldada en el ecosistema científico es el estilo de documentación de NumPy

Retorno - return

Si lo que buscamos es escribir una función que **retorne** una lista con los números de la serie de Fibonacci en lugar de imprimirlos, podemos pensar en el siguiente ejemplo para completar:

[ ]:
```
# -*- coding: utf-8 -*-

def fib2(n):
    """Devuelve una lista conteniendo la serie de Fibonacci hasta n."""

    resultado  # inicializo la lista completar!
    a, b = 0, 1  # asignacion multiple
    while a < n:
        resultado.  # completar!
        a, b =  # completar!
    return # completar!

# Invocamos la función definida.
fib2000 = fib2(2000)
type(fib2000)
```

Scope

Es importante resaltar que **las variables que se crean dentro de las funciones no son accesibles una vez que termina la ejecución de la función**. En cambio, la función si que puede acceder a cosas que se han definido fuera de ella. No obstantes, esto último no constituye una buena práctica de cara a la reproducibilidad, mantenibilidad y testeo de la función.

Argumentos

También es posible definir funciones con un número variable de argumentos. Hay tres formas que pueden ser combinadas:

- Argumentos con valores por omisión.
- Palabras claves como argumentos.
- Listas de argumentos arbitrarios.

Argumentos con valores por omision La forma más útil es especificar un valor por omisión para uno o más argumentos. Esto crea una función que puede ser llamada con menos argumentos que los que permite. Por ejemplo:

[22]:
```
def pedir_confirmacion(prompt, reintentos=4, recordatorio="Por favor, intente␣
↪nuevamente!"):
    while True:
        ok = input(prompt)
        if ok in ("s", "S", "si", "Si", "SI", "sI"):  # contemplando todos los␣
↪casos, "in" palabra resevada para probar si una secuencia contiene o no un␣
↪determinado valor.
            return True
        if ok in ("n", "N", "no", "No", "NO", "nO"):  # contemplando todos los␣
↪casos
            return False
        reintentos -= 1
        if reintentos < 0:
            raise ValueError("respuesta de usuario inválida")
        print (recordatorio)
```

[ ]:
```
# pasando solo el argumento obligatorio

pedir_confirmacion("¿Realmente queres salir?")
```

[ ]:
```
# pasando uno de los argumento opcionales

pedir_confirmacion("¿Sobreescribir el archivo?", 2)
```

[ ]:
```
# pasando todos los argumentos

pedir_confirmacion("¿Sobreescribir el archivo?", 2, "Vamos, solo si o no!")
```

Palabras claves como argumentos Las funciones también puede ser llamadas usando argumentos con palabras claves (o argumentos nombrados) de la forma **keyword = value**. Por ejemplo, la siguiente función:

[23]:
```
def loro(tension, estado='muerto', accion='explotar', tipo='Azul Nordico'):
    print("-- Este loro no va a", accion, end=' ')
    print("si le aplicás", tension, "voltios.")
    print("-- Gran plumaje tiene el", tipo)
```

```
    print("-- Está", estado, "!")
```

Acepta un argumento obligatorio (tension) y tres argumentos opcionales (estado, accion, y tipo). Esta función puede llamarse de cualquiera de las siguientes maneras:

[24]:
```
loro(1000)  # 1 argumento posicional
```

```
-- Este loro no va a explotar si le aplicás 1000 voltios.
-- Gran plumaje tiene el Azul Nordico
-- Está muerto !
```

[25]:
```
loro(tension=1000)  # 1 argumento nombrado, palabra clave
```

```
-- Este loro no va a explotar si le aplicás 1000 voltios.
-- Gran plumaje tiene el Azul Nordico
-- Está muerto !
```

[26]:
```
loro(tension=1000000, accion='BOOOOOM')  # 2 argumentos nombrados
```

```
-- Este loro no va a BOOOOOM si le aplicás 1000000 voltios.
-- Gran plumaje tiene el Azul Nordico
-- Está muerto !
```

[27]:
```
loro(accion='BOOOOOM', tension=1000000) # 2 argumentos nombrados, sin orden
```

```
-- Este loro no va a BOOOOOM si le aplicás 1000000 voltios.
-- Gran plumaje tiene el Azul Nordico
-- Está muerto !
```

Referencias

- *Numpy User Guide*, https://www.numpy.org/
- Scott Shell, *An introduction to Numpy and Scipy*, 2014.
- G. Van Rossum. El tutorial de Python. PyAr http://docs.python.org.ar/tutorial/
- Datos y variables. 2009
- Nogueras, Guillem. Introducción informal a Matlab y Octave. Universidad Politecnica de Madrid, 2007
- Massimo Di Pierro. Web2py - Manual de Referencia. Extraido de www.web2py.com, 2018.
- Eugenia Bahit. Python para principiante. Extraido de Libros Web, 2018.
- INTI - Electrónica e Informática, UT Comunicaciones. Introducción al Procesamiento Digital de Señales, 2017.

Licencia

- mas numpy arrays
- control de flujo
- funciones