



- [El 'ndarray' de NumPy: un Array Multidimensional](#)
  - [Creando ndarrays](#)
  - [Copias y vistas](#)
  - [Aritmética con NumPy Arrays](#)
  - [Álgebra Lineal](#)
  - [Funciones Universales](#)
  - [Agregar y eliminar elementos](#)
  - [Manipulación de dimensiones](#)
  - [Indexación y Partición básicas](#)
  - [Indexación avanzada](#)
  - [Concatenación y división de arrays](#)
  - [Recorrido sobre NumPy Arrays](#)
  - [Programación orientada a Arrays](#)
    - [Expresando lógica condicional como operaciones sobre Arrays](#)
    - [Métodos Estadísticos y Matemáticos](#)
    - [Métodos para Arrays booleanos](#)
    - [Ordenación](#)
    - [Unique y otras lógicas de conjuntos](#)
  - [Salvado y recuperación de Arrays en disco](#)
  - [Generación de número pseudoaleatorios](#)
  - [Arrays Estructurados \(registros\)](#)
  - [Datetimes y Timedeltas](#)
  - [Datos faltantes](#)

---

## NumPy: Computación con Matrices

NumPy, abreviatura de *Numerical Python*, es uno de los paquetes fundamentales más importantes para la computación numérica en Python. La mayoría de los paquetes computacionales que proporcionan funcionalidad científica utilizan los objetos de matriz de NumPy como *lingua franca* para el intercambio de datos.

Es necesario importar la librería `NumPy` para poder utilizarla:

```
In [1]: # importación de la librería, el alias es 'np' por convención
import numpy as np
np.__version__
```

```
Out[1]: '1.18.5'
```

Algunos de los componentes de NumPy son:

- `ndarray`, una matriz multidimensional eficiente que proporciona una arquitectura rápida orientada a matrices y operaciones aritméticas.
- Funciones matemáticas para operaciones rápidas sobre matrices completas de datos sin tener que escribir bucles.
- Herramientas para leer/escribir datos de matrices en el disco y trabajar con archivos asignados en memoria.
- Álgebra lineal, generación de números aleatorios y capacidades de transformaciones de Fourier.
- Una API de C para conectar NumPy con bibliotecas escritas en C, C++ o FORTRAN.

Si bien NumPy por sí solo no proporciona funciones de modelado o científicas, comprender las matrices de NumPy y la computación orientada a matrices es esencial para usar herramientas con semántica orientada a matrices, como Pandas, mucho más efectivamente.

Las principales áreas de funcionalidad para la mayoría de las aplicaciones de análisis de datos son:

- Operaciones rápidas sobre matrices vectorizadas para la recopilación y limpieza de datos, definición de subconjuntos y filtros, transformación y cualquier otro tipo de cálculo.
- Algoritmos comunes sobre matrices como clasificación y operaciones de conjuntos.
- Estadística descriptiva eficiente con agregación/resumen de datos
- Alineación de datos y manipulación de datos relacionales para fusionar y unir conjuntos de datos heterogéneos.
- Expresar la lógica condicional como expresiones matriciales en lugar de bucles con ramas `if-elif-else`
- Manipulaciones de datos a nivel de grupo (agregación, transformación, aplicación de funciones)

Una de las razones por las que NumPy es tan importante para los cálculos numéricos en Python es porque está diseñado para la eficiencia en grandes conjuntos de datos. Hay un número de razones para esto:

- NumPy almacena internamente los datos en un bloque de memoria contiguo, independientemente de otros objetos Python incorporados. La biblioteca de algoritmos de NumPy escrita en el lenguaje C puede operar en esta memoria sin ningún tipo de verificación de tipo u otra sobrecarga. Las matrices NumPy también usan mucha menos memoria que las secuencias Python incorporadas.
- Las operaciones NumPy realizan cálculos complejos en matrices completas sin la necesidad de bucles Python.

Para dar una idea de la diferencia de rendimiento, consideremos una matriz NumPy de enteros y la lista equivalente de Python:

```
In [2]: # import numpy as np
my_arr = np.arange(10000000)

# multiplicamos cada matriz por 2 y obtenemos los tiempos
%time for _ in range(10): my_arr2 = my_arr * 2

CPU times: user 162 ms, sys: 66.1 ms, total: 228 ms
Wall time: 228 ms
```

```
In [3]: my_list = list(range(10000000))
%time for _ in range(10): my_list2 = [x * 2 for x in my_list]

CPU times: user 6.38 s, sys: 1.49 s, total: 7.87 s
Wall time: 7.87 s
```

La diferencia clave entre una matriz NumPy y una lista Python es que las matrices están diseñadas para manejar operaciones vectorizadas mientras que una lista de Python está diseñada para realizar operaciones *individuales* sobre cada uno de sus componentes.

```
In [4]: # Generamos algunos datos aleatorios
data = np.random.randn(2, 3) # método `randn` del subpaquete 'random'
data

Out[4]: array([[ -0.18932497,  1.53374757, -0.24261478],
               [ 0.28517945, -0.1175744 ,  0.30773107]])
```

```
In [5]: # multiplicamos por 10 el array
data * 10
```

```
Out[5]: array([[ -1.89324971,  15.33747565, -2.42614781],
               [  2.85179454, -1.17574402,  3.07731067]])
```

```
In [6]: # sumamos el array con el mismo
data + data
```

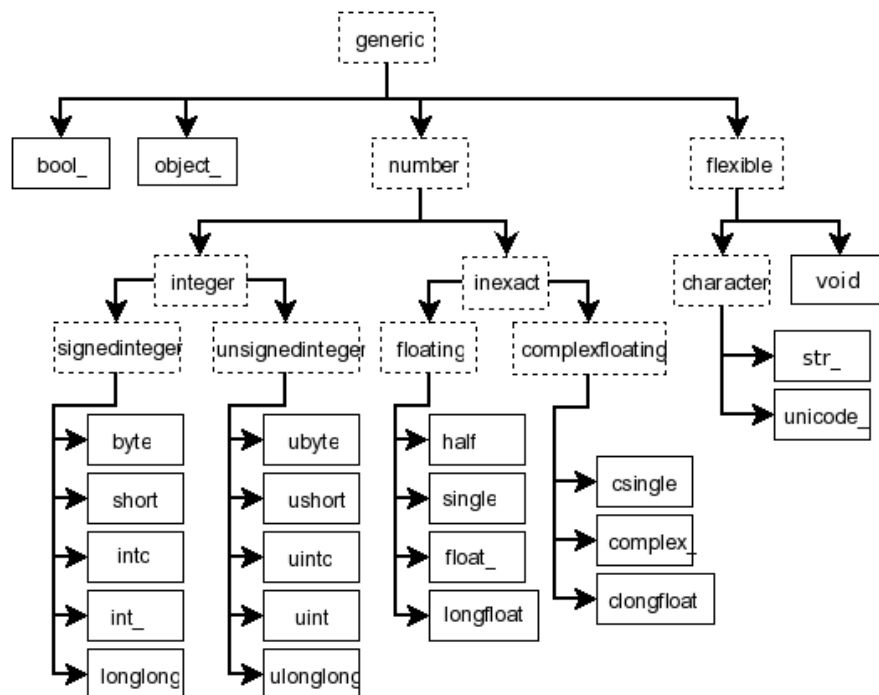
```
Out[6]: array([[ -0.37864994,  3.06749513, -0.48522956],
               [  0.57035891, -0.2351488 ,  0.61546213]])
```

Los algoritmos basados en NumPy son generalmente de 10 a 100 veces más rápidos (o más) que sus equivalentes puros de Python y utilizan significativamente menos memoria.

## Tipos de datos NumPy

Python define solo un tipo de una clase de datos particular (solo hay un tipo entero, un tipo de punto flotante, etc.). Esto puede ser conveniente en aplicaciones que no necesitan preocuparse por todas las formas en que se pueden representar los datos en una computadora. Para la computación científica, sin embargo, a menudo se necesita más control.

En NumPy, hay 24 nuevos tipos fundamentales de Python para describir diferentes tipos de escalares. Estos descriptores de tipo se basan principalmente en los tipos disponibles en el lenguaje C en el que está escrito CPython, con varios tipos adicionales compatibles con los tipos de Python.



## El ndarray de NumPy: un Array Multidimensional

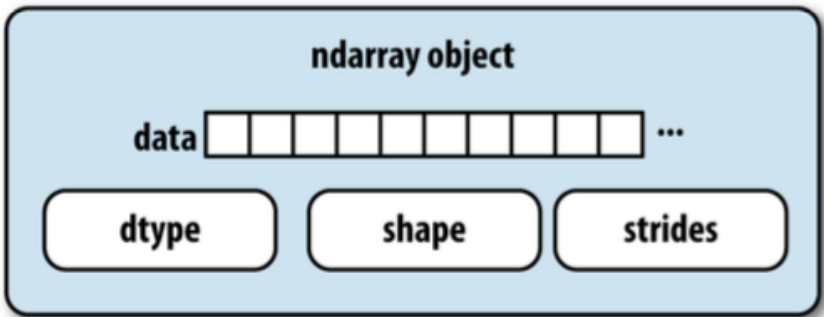
Una de las características clave de NumPy es su objeto de matriz N-dimensional, o `ndarray`, que es un contenedor rápido y flexible para grandes conjuntos de datos en Python. Las matrices le permiten realizar operaciones matemáticas en bloques enteros de datos utilizando una sintaxis similar a las operaciones equivalentes entre elementos escalares. Un `ndarray` describe una colección de "elementos" del mismo tipo. Los elementos se pueden indexar utilizando, por ejemplo, N enteros.

**Todos los ndarrays son homogéneos**, por lo que cada elemento ocupa el mismo tamaño de bloque de memoria, y todos los bloques se interpretan de la misma manera. La forma en que debe interpretarse cada elemento de la matriz se especifica mediante un objeto de tipo de datos independiente, uno de los cuales está asociado con cada matriz. Además de los tipos básicos (enteros, flotantes, etc.), los objetos de tipo de datos también pueden representar estructuras de datos.

Un elemento extraído de una matriz, por ejemplo, mediante la indexación, se representa mediante un objeto Python cuyo tipo es uno de los tipos escalares de matriz integrados en NumPy. Los escalares de matriz permiten una fácil manipulación de arreglos de datos también más complicados.

El NumPy `ndarray` proporciona un medio para interpretar un bloque de datos homogéneos (ya sea contiguos o no) como una matriz multidimensional. Parte de lo que hace que un `ndarray` sea flexible es ser una vista sobre un bloque de datos, no es sólo una porción de memoria sino que contiene información sobre los datos, cómo localizar un elemento y cómo interpretar un elemento. Más precisamente, el `ndarray` internamente consiste en lo siguiente:

- Un puntero a los datos ( `data` ), es decir, un bloque de datos en la RAM o en un archivo asignado en memoria
- El tipo de dato ( `dtype` ), que describe celdas de valor de tamaño fijo en la matriz
- Una tupla que indica la forma de la matriz ( `shape` )
- Una tupla de pasos ( `strides` ), que son el número de bytes que se deben omitir en la memoria para ir al siguiente elemento. Si el valor es (10,1), hace falta un byte para llegar a la siguiente columna y 10 bytes para ubicar la siguiente fila.
- Indicadores ( `flags` ) que definen entre otros elementos si se nos permite modificar la matriz, si el diseño de la memoria es formato C o Fortran, y así sucesivamente.



Resumiendo, podemos ver un `ndarray` como un contenedor genérico multidimensional para **datos homogéneos**; es decir, todos los elementos deben ser del mismo tipo. Indexado por una tupla de valores enteros positivos. En NumPy las dimensiones se denominan `ejes`. Los atributos más importantes de un objeto `ndarray` son:

Atributos	Descripción
<code>ndarray.flags</code>	Información sobre el diseño de memoria de la matriz
<code>ndarray.shape</code>	Tupla de dimensiones de matriz
<code>ndarray.strides</code>	Tupla de bytes para pasar en cada dimensión al atravesar una matriz
<code>ndarray.ndim</code>	Número de dimensiones de la matriz
<code>ndarray.data</code>	El objeto de búfer de Python que apunta al inicio de los datos de la matriz
<code>ndarray.dtype</code>	Objeto que describe el tipo de los elementos en la matriz
<code>ndarray.size</code>	Número de elementos en la matriz

Atributos	Descripción
ndarray.itemsize	Longitud de un elemento de matriz en bytes
ndarray.nbytes	Total de bytes consumidos por los elementos de la matriz

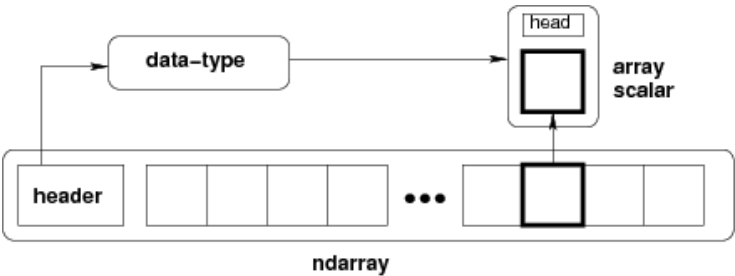
```
In [7]: f'{data.ndim} dimensiones {data.shape}, un tamaño de {data.size} elementos de tipo {data.dtype}'

Out[7]: '2 dimensiones (2, 3), un tamaño de 6 elementos de tipo float64'
```

Diseño de memoria interna de un 'ndarray'

Una instancia de la clase ndarray consiste en un segmento unidimensional contiguo de la memoria de la computadora (que pertenece a la matriz o por algún otro objeto), combinado con un esquema de indexación que asigna N enteros a la ubicación de un elemento en el bloque. Los rangos en los que pueden variar los índices están especificados por la forma de la matriz. La cantidad de bytes que toma cada elemento y cómo se interpretan los bytes se define por el objeto de tipo de datos asociado con la matriz.

Un segmento de la memoria es inherentemente unidimensional, y hay muchos esquemas diferentes para organizar los elementos de una matriz N-dimensional en un bloque unidimensional. NumPy es flexible, y los objetos ndarray pueden adaptarse a cualquier esquema de indexación por pasos.



Creando ndarrays

La forma más fácil de crear una matriz es usar la función array que acepta cualquier objeto similar a una secuencia (listas, tuplas y otras matrices) y produce una nueva matriz NumPy que contiene los datos pasados.

```
In [8]: # matriz a partir de una lista
data1 = [6, 7.5, 8, 0, 1]
arr1 = np.array(data1)
arr1

Out[8]: array([6. , 7.5, 8. , 0. , 1. ])

In [9]: # matriz a partir de una tupla
data1 = (6, 7.5, 8, 0, 1)
arr1 = np.array(data1)
arr1

Out[9]: array([6. , 7.5, 8. , 0. , 1. ])
```

Las secuencias anidadas, como una lista de listas de igual longitud, se convertirán en una matriz multidimensional:

```
In [10]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
arr2 = np.array(data2)
arr2
```

```
Out[10]: array([[1, 2, 3, 4],
               [5, 6, 7, 8]])
```

```
In [11]: print('{0} ejes con tamaños {1}'.format(arr2.ndim, arr2.shape))

2 ejes con tamaños (2, 4)
```

El tipo de los datos de un `ndarray` puede ser especificado en tiempo de creación configurando el argumento `dtype`. Algunos de los tipos de tipos más comunes son: 'float', 'int', 'bool', 'str' y 'object', pero para controlar las asignaciones de memoria, se puede optar por usar 'float16', 'float32', 'float64', 'float128', 'int8', 'int16', 'int32', 'int64', 'uint8', 'uint16', 'uint32' o 'uint64'. Por defecto, el `dtype` de la matriz creada es `float64`, pero el sistema intenta ajustar al tipo más cercano:

```
In [12]: print('Tipo arr1 {0}; tipo arr2 {1}'.format(arr1.dtype, arr2.dtype))

Tipo arr1 float64; tipo arr2 int64
```

Independientemente del tipo original de los datos de una matriz, éste se podrá cambiar utilizando el método `astype`.

```
In [13]: arr3 = np.array( [ [1,2], [3,4] ], dtype='int16' )
arr3
```

```
Out[13]: array([[1, 2],
               [3, 4]], dtype=int16)
```

```
In [14]: # cambio de tipo 'int16' a 'complex'
arr3.astype('complex')
```

```
Out[14]: array([[1.+0.j, 2.+0.j],
               [3.+0.j, 4.+0.j]])
```

A menudo, los elementos de una matriz son originalmente desconocidos, pero su tamaño es conocido. NumPy ofrece varias funciones para crear matrices con contenido inicial que minimizan la necesidad de matrices dinámicas, una operación muy costosa. La función `zeros` crea una matriz llena de ceros, la función `ones` crea una matriz llena de unos y la función `empty` crea una matriz cuyo contenido inicial es aleatorio y depende del estado de la memoria.

```
In [15]: np.zeros((3, 6))
```

```
Out[15]: array([[0., 0., 0., 0., 0., 0.],
               [0., 0., 0., 0., 0., 0.],
               [0., 0., 0., 0., 0., 0.]])
```

```
In [16]: np.ones((2, 3))
```

```
Out[16]: array([[1., 1., 1.],
               [1., 1., 1.]])
```

```
In [17]: np.empty((2, 3, 2), dtype=np.int16)

Out[17]: array([[[ 0, 0],
 [ 0, 0],
 [ 0, 0]],

 [[ 0, 0],
 [8344, 1760],
 [2044, -20480]]], dtype=int16)
```

No se puede asumir que `np.empty` devolverá una matriz de ceros. En algunos casos, puede devolver valores "basura" sin inicializar.

Como se ha indicado anteriormente, a diferencia de las listas todos los elementos de una matriz Numpy debe ser del mismo tipo de datos. Cuando se proporcionan diferentes tipos de datos en la creación de las matrices, el sistema realiza una conversión de tipos dinámica si no se indica lo contrario. Si la conversión fallara por algún motivo (como una cadena que no se puede convertir a `float64`), se generará el correspondiente `ValueError`.

```
In [18]: # valores numéricos y booleanos se convierten automáticamente a tipo float
arr1 = np.array([111, 2.3, True, False, False])
print(arr1)
arr1.dtype

[111.  2.3  1.  0.  0.]
```

```
Out[18]: dtype('float64')
```

```
In [19]: # valores numéricos y cadenas se convierten automáticamente a tipo cadena
arr1 = np.array([111, 2.3, 'python', 'abc'])
arr1
```

```
Out[19]: array(['111', '2.3', 'python', 'abc'], dtype='<U32')
```

```
In [20]: # valores numéricos, booleanos y cadenas se convierten automáticamente a tipo c
adena
arr1 = np.array([111, True, 'abc'])
arr1
```

```
Out[20]: array(['111', 'True', 'abc'], dtype='<U21')
```

Si no se está seguro de qué tipo de datos tendrá una matriz o si se desea mantener caracteres y números en la misma matriz, se puede configurar el tipo de `dtype` como `object`.

```
In [21]: # valores numéricos, booleanos y cadenas se convierten a objetos manteniendo su
naturaleza
arr1_obj = np.array([1, 'a', True], dtype='object')
arr1_obj
```

```
Out[21]: array([1, 'a', True], dtype=object)
```

Un `array` Numpy siempre se puede convertir en una `lista` Python utilizando el método `tolist()`:

```
In [22]: arr1_obj.tolist()
```

```
Out[22]: [1, 'a', True]
```

Para resumir, las principales diferencias con las listas de python son:

- Las matrices admiten operaciones vectorizadas, mientras que las listas no lo hacen.
- Una vez que se crea una matriz, no puede cambiar su tamaño. Se tendrá que crear una nueva matriz o sobrescribir la existente.
- Cada matriz tiene uno y solo un `dtype`. Todos los elementos deben ser de ese tipo.
- Una matriz Numpy equivalente ocupa mucho menos espacio que una lista de listas de Python.

Para crear secuencias de números, NumPy proporciona la función `arange` análoga a la función `range` de Python que devuelve matrices en lugar de listas.

```
In [23]: np.arange(15)
```

```
Out[23]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Vinculado al método `arange` suele aparecer el uso del método `reshape` que permite cambiar las dimensiones de una matriz:

```
In [24]: np.arange(15).reshape(3,5)
```

```
Out[24]: array([[ 0,  1,  2,  3,  4],
               [ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14]])
```

```
In [25]: # de 0 a 9 con paso 2
np.arange(0, 10, 2)
```

```
Out[25]: array([0, 2, 4, 6, 8])
```

```
In [26]: # de 10 a 1, en orden decreciente
np.arange(10, 0, -1)
```

```
Out[26]: array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
```

Con `arange` se pueden establecer las posiciones de inicio y final, pero no se puede indicar el número de elementos del array, para ello habría que calcular manualmente el valor de paso apropiado. Para ese tipo de operaciones está el método `linspace`:

```
In [27]: # 10 números entre el 1 y el 50
np.linspace(start=1, stop=50, num=10, dtype=int)
```

```
Out[27]: array([ 1,  6, 11, 17, 22, 28, 33, 39, 44, 50])
```

Similar a `linspace`, pero utilizando una escala logarítmica está `logspace`:

```
In [28]: # Limita el número de dígitos decimales a 2
np.set_printoptions(precision=2)

# 10 números entre 10^1 y 10^50
np.logspace(start=1, stop=50, num=10, base=10)
```

```
Out[28]: array([1.00e+01, 2.78e+06, 7.74e+11, 2.15e+17, 5.99e+22, 1.67e+28,
               4.64e+33, 1.29e+39, 3.59e+44, 1.00e+50])
```

Por último para crear secuencias repetidas de valores en matrices también están los métodos `tile` y `repeat`:



```
In [29]: list1 = [1,2,3]
# repite todo 'list1' dos veces
np.tile(list1, 2)
```

```
Out[29]: array([1, 2, 3, 1, 2, 3])
```

```
In [30]: # repite cada elemento de 'list1' 2 veces
np.repeat(list1, 2)
```

```
Out[30]: array([1, 1, 2, 2, 3, 3])
```

Las matrices multidimensionales admiten el parámetro `axis` :

```
In [31]: list1 = [[2, 3], [5, 6], [7, 9]]
np.repeat(list1, repeats=3, axis=1)
```

```
Out[31]: array([[2, 2, 2, 3, 3, 3],
               [5, 5, 5, 6, 6, 6],
               [7, 7, 7, 9, 9, 9]])
```

La siguiente tabla muestra algunas de las funciones que permiten crear un `ndarray` :

Función	Descripción
<code>array</code>	Convierte los datos de entrada (lista, tupla, matriz u otro tipo de secuencia) en un <code>ndarray</code> inferiendo un <code>dtype</code> o especificando explícitamente un <code>dtype</code> ; copia los datos de entrada por defecto.
<code>asarray</code>	Convierte la entrada a <code>ndarray</code> , pero no la copia si la entrada ya es una <code>ndarray</code>
<code>arange</code>	Al igual que <code>range</code> , pero devuelve un <code>ndarray</code> en lugar de una lista
<code>ones</code> , <code>ones_like</code>	Produce una matriz de todos los <code>1s</code> con la forma y el tipo dados; <code>ones_like</code> toma otra matriz y produce una matriz de unidades de la misma forma y tipo
<code>zeros</code> , <code>zeros_like</code>	Al igual que <code>ones</code> y <code>ones_like</code> pero produce matrices de <code>0s</code>
<code>empty</code> , <code>empty_like</code>	Al igual que <code>ones</code> y <code>ones_like</code> pero no se inicializan los valores
<code>full</code> , <code>full_like</code>	Genera una matriz de la forma y el tipo dados con todos los valores inicializados con el "valor de relleno". <code>full_like</code> toma otra matriz y produce una matriz completa de la misma forma y tipo.
<code>eye</code> , <code>identity</code>	Crea una matriz de identidad cuadrada $N \times N$ (1's en la diagonal y 0's en otra parte)

### Copias y vistas

Al operar y manipular matrices, sus datos a veces se copian en una nueva matriz y otras veces no. Esto es a menudo una fuente de confusión para los principiantes. Hay tres casos:

**1.- Sin copia en absoluto** Las asignaciones simples no hacen copia de los objetos matriz o de sus datos.

```
In [32]: a = np.arange(12)
b = a    # no se crea ningún objeto nuevo
b is a   # a y b son dos nombres para el mismo objeto
```

```
Out[32]: True
```

```
In [33]: id(b) == id(a) # a y b comparten la id del objeto
```

```
Out[33]: True
```

```
In [34]: b.shape = 3,4 # cambia el shape de b y de a
         a.shape
```

```
Out[34]: (3, 4)
```

**2.- Vista o Copia superficial** Diferentes objetos matriz pueden compartir los mismos datos. Tal y como se ha indicado anteriormente, la segmentación siempre devuelve vistas a la matriz original. Adicionalmente el método `view` crea un nuevo objeto matriz que utiliza los mismos datos que la matriz original.

```
In [35]: c = a.view() # c no es a
         c is a
```

```
Out[35]: False
```

```
In [36]: c.base is a # c es una vista de los datos de a
```

```
Out[36]: True
```

```
In [37]: c.flags.owndata # c no tiene sus propios datos
```

```
Out[37]: False
```

```
In [38]: c.shape = 2,6 # el shape de a no cambia
         a.shape
```

```
Out[38]: (3, 4)
```

```
In [39]: c[0,4] = 1234 # los datos de a cambian
         a
```

```
Out[39]: array([[ 0,  1,  2,  3],
                [1234,  5,  6,  7],
                [ 8,  9, 10, 11]])
```

**3.- Copia profunda** El método `copy` hace una copia completa de la matriz y sus datos.

```
In [40]: d = a.copy() # se crea un nuevo objeto con nuevos datos
         d is a
```

```
Out[40]: False
```

```
In [41]: d.base is a # d no comparte nada con a
```

```
Out[41]: False
```

```
In [42]: id(d) == id(a) # a y d tienen id's diferentes
```

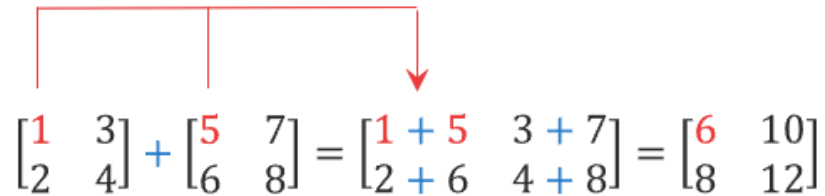
```
Out[42]: False
```

```
In [43]: d[0,0] = 9999
         a
```

```
Out[43]: array([[ 0,  1,  2,  3],
                [1234,  5,  6,  7],
                [ 8,  9, 10, 11]])
```

## Aritmética con NumPy Arrays

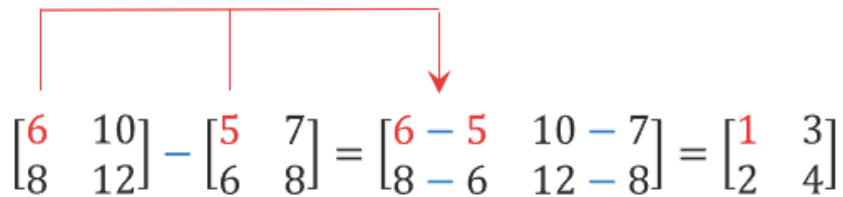
Las matrices `ndarray` son importantes porque permiten expresar operaciones por lotes en los datos sin escribir bucles. Los usuarios NumPy llaman a esta propiedad *vectorización*. Cualquier operación aritmética entre **matrices de igual tamaño** aplica la operación de manera elemental:



$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 1+5 & 3+7 \\ 2+6 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 10 \\ 8 & 12 \end{bmatrix}$$

```
In [44]: arr1 = np.array([[1., 3.], [2., 4.]])
arr2 = np.array([[5., 7.], [6., 8.]])
arr3 = arr1 + arr2
arr3
```

```
Out[44]: array([[ 6., 10.],
               [ 8., 12.]])
```



$$\begin{bmatrix} 6 & 10 \\ 8 & 12 \end{bmatrix} - \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 6-5 & 10-7 \\ 8-6 & 12-8 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

```
In [45]: arr3 - arr2
```

```
Out[45]: array([[1., 3.],
               [2., 4.]])
```

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} * \begin{bmatrix} 7 & 9 & 11 \\ 8 & 10 & 12 \end{bmatrix} = \begin{bmatrix} 1 \times 7 & 3 \times 9 & 5 \times 11 \\ 2 \times 8 & 4 \times 10 & 6 \times 12 \end{bmatrix} = \begin{bmatrix} 7 & 27 & 55 \\ 16 & 40 & 72 \end{bmatrix}$$

```
In [46]: arr1 = np.array([[1., 3., 5.], [2., 4., 6.]])
arr2 = np.array([[7., 9., 11.], [8., 10., 12.]])
arr3 = arr1 * arr2
```

$$\begin{bmatrix} 7 & 27 & 55 \\ 16 & 40 & 72 \end{bmatrix} / \begin{bmatrix} 7 & 9 & 11 \\ 8 & 10 & 12 \end{bmatrix} = \begin{bmatrix} 7/7 & 27/9 & 55/11 \\ 16/8 & 40/10 & 72/12 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

```
In [47]: arr3 / arr2
```

```
Out[47]: array([[1., 3., 5.],
               [2., 4., 6.]])
```

```
In [48]: arr1 ** 0.5
```

```
Out[48]: array([[1.   , 1.73, 2.24],
               [1.41, 2.   , 2.45]])
```

```
In [49]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
arr2
```

```
Out[49]: array([[ 0.,  4.,  1.],
               [ 7.,  2., 12.]])
```

```
In [50]: arr2 > arr1
```

```
Out[50]: array([[False,  True, False],
               [ True, False,  True]])
```

Algunas operaciones, como `+=` y `*=`, modifican la matriz actual en lugar de crear una nueva.

```
In [51]: arr2 += 3
arr2
```

```
Out[51]: array([[ 3.,  7.,  4.],
               [10.,  5., 15.]])
```

### Broadcasting (difusión) en matrices

NumPy tiene un conjunto de reglas para tratar con matrices que tienen diferentes formas, que se aplican cada vez que las funciones toman múltiples operandos que se combinan de forma elemental. Esta característica se llama `Broadcasting`.

En el siguiente ejemplo vemos cómo en una operación aritmética de una matriz de 1D con un escalar aplica esta característica:

```
In [6]: arr1d = np.array([0, 1, 2])
arr1d + 5
```

```
Out[6]: array([5, 6, 7])
```

Podemos pensar en una operación que extiende o duplica el valor 5 en la matriz `[5, 5, 5]` y suma los resultados. La ventaja de la difusión de NumPy es que esta duplicación de valores no ocurre realmente. Esto es aplicable no sólo a operaciones con escalares.

```
In [52]: arr2d = np.arange(12.).reshape((3, 4))
arr2d
```

```
Out[52]: array([[ 0.,  1.,  2.,  3.],
               [ 4.,  5.,  6.,  7.],
               [ 8.,  9., 10., 11.]])
```

```
In [53]: arr1d = arr2d[0]
```

```
In [54]: arr2d - arr1d
```

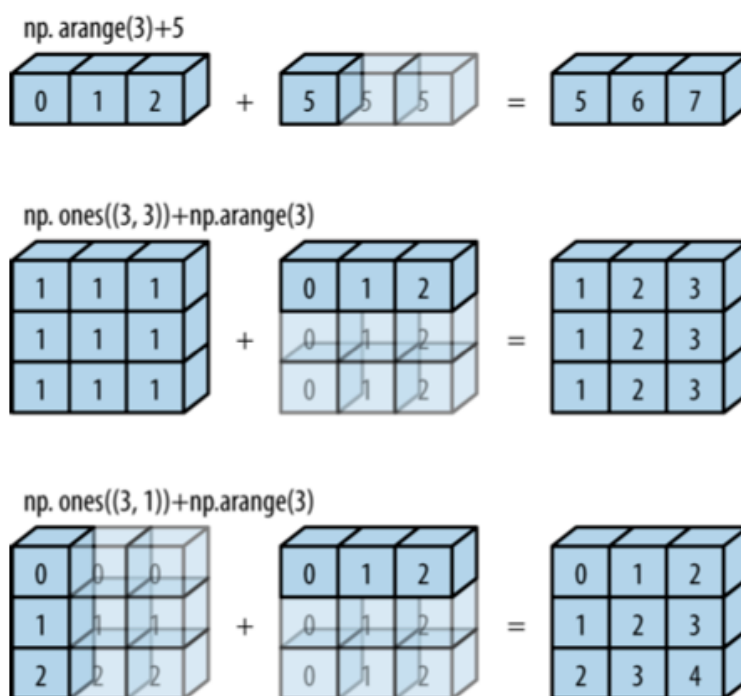
```
Out[54]: array([[0., 0., 0., 0.],
               [4., 4., 4., 4.],
               [8., 8., 8., 8.]])
```

Aquí, la matriz unidimensional `arr1d` se estira, o se difunde, a través de la segunda dimensión para que coincida con la forma de `arr2d`.

```
In [4]: a = np.arange(3)
        b = np.arange(3)[: , np.newaxis]
        print(a)
        print(b)
        a + b
```

```
[0 1 2]
[[0]
 [1]
 [2]]
```

```
Out[4]: array([[0, 1, 2],
               [1, 2, 3],
               [2, 3, 4]])
```



Las reglas de la difusión son:

- Regla 1: Si las dos matrices difieren en su número de dimensiones, la forma de la que tiene menos dimensiones se rellena con unos en su lado principal (izquierdo).
- Regla 2: En dos matrices con igual número de dimensiones, si no coincide el tamaño en alguna dimensión, la matriz con tamaño igual a 1 en la dimensión no coincidente se estira para que coincida con el tamaño de la otra matriz en la misma dimensión.
- Regla 3: En dos matrices con igual número de dimensiones, si en alguna dimensión los tamaños no concuerdan y ninguno es igual a 1, se genera un error.

**Nota:** Si se quiere hacer compatibles dos matrices y para ello se requiere el relleno del lado derecho, se debe hacer cambiando la forma de la matriz.

```
In [9]: arr2d = np.ones((3, 2))
arr1d = np.arange(3)
arr2d + arr1d
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-9-cccb5b64d4e5> in <module>
      1 arr2d = np.ones((3, 2))
      2 arr1d = np.arange(3)
----> 3 arr2d + arr1d

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

```
In [10]: arr1d[:, np.newaxis].shape
```

```
Out[10]: (3, 1)
```

```
In [12]: arr2d + arr1d[:, np.newaxis]
```

```
Out[12]: array([[1., 1.],
               [2., 2.],
               [3., 3.]])
```

## Álgebra Lineal

El álgebra lineal, como el producto de matrices, las descomposiciones, los determinantes y otras matemáticas para matrices cuadradas, es una parte importante de cualquier biblioteca de matrices. A diferencia de algunos lenguajes como MATLAB, multiplicar dos matrices bidimensionales con `*` es una multiplicación de elementos en lugar de un producto de matrices. Numpy proporciona la función y el método `dot` para la multiplicación de matrices. También se puede realizar utilizando el operador `@` (en python >= 3.5):

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} @ \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 1 \times 7 + 3 \times 9 + 5 \times 11 & 1 \times 8 + 3 \times 10 + 5 \times 12 \\ 2 \times 7 + 4 \times 9 + 6 \times 11 & 2 \times 8 + 4 \times 10 + 6 \times 12 \end{bmatrix}$$

$$= \begin{bmatrix} 89 & 98 \\ 116 & 128 \end{bmatrix}$$

La multiplicación de matrices a nivel de elementos necesita que los mismos tengan las mismas dimensiones. En el caso del producto de matrices propio del álgebra lineal, el número de columnas en la matriz izquierda debe ser igual al número de filas en la matriz derecha. Cuando una matriz  $p \times m$  se multiplica por otra de tamaño  $m \times q$ , la matriz resultante tiene forma  $p \times q$ .

```
In [55]: A = np.array( [[1,3,5], [2,4,6]] )
B = np.array( [[7,8], [9,10], [11,12]] )
# producto de matrices con operador
A @ B
```

```
Out[55]: array([[ 89,  98],
               [116, 128]])
```

```
In [56]: # otro producto de matrices con método
A.dot(B)
```

```
Out[56]: array([[ 89,  98],
               [116, 128]])
```

```
In [57]: # otro producto de matrices con función
         np.dot(A, B)
```

```
Out[57]: array([[ 89,  98],
               [116, 128]])
```

## Matriz transpuesta

La matriz transpuesta de una dada es una matriz en la que las filas son reemplazadas por las columnas. La función de transposición `transpose` de Numpy se utiliza para transponer una matriz  $A$  y transformarla en una matriz transpuesta  $A^T$ . Además toda matriz Numpy tiene la propiedad `T` que devuelve su matriz transpuesta.

$$x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad y = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

```
In [58]: X = np.array([[1,2,3], [4,5,6], [7,8,9]])
         X.T
```

```
Out[58]: array([[1, 4, 7],
               [2, 5, 8],
               [3, 6, 9]])
```

```
In [59]: np.transpose(X)
```

```
Out[59]: array([[1, 4, 7],
               [2, 5, 8],
               [3, 6, 9]])
```

## Matriz inversa

La matriz inversa de una matriz  $A$  es tal que cuando se multiplica  $A$  por su inversa (de notada normalmente como  $A^{-1}$ ) se obtiene la matriz de identidad ( $I$ ). La matriz inversa no siempre existe, solo se pueden invertir matrices cuadradas no singulares. Una matriz identidad o *unidad de orden  $n$*  es una matriz cuadrada donde todos sus elementos son ceros (0) menos los elementos de la diagonal principal que son unos (1).

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} @ \begin{bmatrix} -2 & 1 \\ 1.5 & -0.5 \end{bmatrix} = \begin{bmatrix} -2 + 3 & 1 - 1 \\ -6 + 6 & 3 - 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$\uparrow$   
 $A$

$\uparrow$   
 $A^{-1}$

$\uparrow$   
 $I$

El módulo `numpy.linalg` tiene un conjunto estándar operaciones de álgebra lineal sobre matrices como descomposiciones matriciales, matrices inversas y obtención del determinante, que se implementan bajo las mismas bibliotecas de álgebra lineal estándar de la industria que se utilizan en otros idiomas como MATLAB y R, tales como BLAS, LAPACK, y MKL (Math Kernel Library de Intel) dependiendo de la compilación de NumPy:

```
In [60]: A = np.array([[1., 2.], [3., 4.]])
         A
Out[60]: array([[1., 2.],
               [3., 4.]])

In [61]: Ainv = np.linalg.inv(A)
         Ainv
Out[61]: array([[-2. ,  1. ],
               [ 1.5, -0.5]])

In [62]: np.dot(A, Ainv)
Out[62]: array([[1.00e+00,  1.11e-16],
               [0.00e+00,  1.00e+00]])

In [63]: np.round(np.dot(A, Ainv), 2)
Out[63]: array([[1.,  0.],
               [0.,  1.]])

In [64]: # np.allclose devuelve True si dos matrices son iguales a nivel de elementos de
         ntro de una tolerancia
         np.allclose(np.identity(2), np.dot(A, Ainv))
Out[64]: True
```

La siguiente tabla algunas de las funciones mas comunes de `numpy.linalg` :

Función	Descripción
diag	Devuelve los elementos de la diagonal principal de una matriz cuadrada como una matriz 1D , o convierte una matriz 1D en una matriz cuadrada con ceros en la diagonal principal
trace	Calcula la suma de los elementos de la diagonal principal
det	Calcula el determinante de una matriz
eig	Calcula los auto-valores y los auto-vectores de una matriz cuadrada
inv	Calcula la inversa de una matriz cuadrada
pinv	Calcula el pseudoinverso de Moore-Penrose de una matriz
qr	Calcula la descomposición QR
svd	Calcula el valor singular de descomposición (SVD)
solve	Resuelve el sistema lineal $Ax = b$ para $x$ , donde $A$ es una matriz cuadrada
lstsq	Calcula la solución de mínimos cuadrados para $Ax = b$



## Sistemas de ecuaciones lineales

Supongamos que tenemos un sistema con **m ecuaciones y n incógnitas**:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \dots a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \dots a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + a_{m3}x_3 \dots a_{mn}x_n &= b_m \end{aligned}$$

Pueden darse tres casos:

1. Hay la misma cantidad de ecuaciones que de incógnitas ( $m = n$ ). En este caso, si hay una solución, es única.
2. Hay más ecuaciones que incógnitas ( $m > n$ ). En este caso se puede obtener una solución aproximada por el método de mínimos cuadrados. Este caso se conoce como "sobredeterminado".
3. Hay más incógnitas que ecuaciones ( $m < n$ ). En este caso se pueden obtener una infinidad de soluciones no triviales si  $b_i = 0$ . Este caso se conoce como "subdeterminado".

En el caso de que haya la misma cantidad de ecuaciones que de incógnitas ( $m = n$ ), la solución se puede calcular usando la matriz inversa. Usando la notación matricial:

$$A x = b$$

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mm} \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}$$

$$x = A^{-1}b$$

```
In [65]: A = np.array([[2, 3, 1], [1, 2, 1], [-1, 4, 0]])
          b = np.array([3, 1, -2])
          np.round(np.dot(np.linalg.inv(A),b), 2)
```

```
Out[65]: array([ 2.,  0., -1.])
```

También se puede utilizar la función `numpy.linalg.solve` :

```
In [66]: np.linalg.solve(A, b)
```

```
Out[66]: array([ 2.,  0., -1.])
```

## Funciones Universales

La implementación predeterminada de Python (conocida como CPython) realiza algunas operaciones muy lentamente. Esto se debe en parte a la naturaleza dinámica e interpretada del lenguaje: el hecho de que los tipos son flexibles, por lo que las secuencias de operaciones no se pueden compilar en un código de máquina eficiente como en lenguajes como C y Fortran.

Para muchos tipos de operaciones, NumPy proporciona una interfaz conveniente en este tipo de rutina compilada de tipo estático. Esto se conoce como operación vectorizada. Puede lograr esto simplemente realizando una operación en la matriz, que luego se aplicará a cada elemento. Este enfoque vectorizado está diseñado para empujar el bucle a la capa compilada que subyace a NumPy, lo que lleva a una ejecución mucho más rápida.

Una función universal, o `ufunc`, es una función que realiza operaciones a nivel de elementos/datos en `ndarrays`. Se puede pensar en éstas como envolturas vectorizadas rápidas para funciones simples que toman uno o más valores escalares y producen uno o más resultados escalares. Muchos `ufunc` son simples transformaciones a nivel de elementos, como `sqrt` o `exp`:

```
In [67]: arr = np.arange(10)
         arr
```

```
Out[67]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [68]: np.sqrt(arr)
```

```
Out[68]: array([0.   , 1.   , 1.41, 1.73, 2.   , 2.24, 2.45, 2.65, 2.83, 3.   ])
```

```
In [69]: np.exp(arr)
```

```
Out[69]: array([1.00e+00, 2.72e+00, 7.39e+00, 2.01e+01, 5.46e+01, 1.48e+02,
                4.03e+02, 1.10e+03, 2.98e+03, 8.10e+03])
```

Éstas se conocen como `ufunc` unarias. Otras, como `add` o `maximum`, toman dos matrices (por lo tanto, son binarias) y devuelven una única matriz como resultado:

```
In [70]: x = np.random.randn(8)
         y = np.random.randn(8)
```

```
In [71]: x
```

```
Out[71]: array([ 0.35,  0.69, -2.59,  1.2 ,  0.9 , -0.43,  0.93,  0.06])
```

```
In [72]: y
```

```
Out[72]: array([ 0.99,  1.22,  0.88, -0.07,  1.25, -0.37,  0.26,  1.47])
```

```
In [73]: np.maximum(x, y)
```

```
Out[73]: array([ 0.99,  1.22,  0.88,  1.2 ,  1.25, -0.37,  0.93,  1.47])
```

Aunque no es común, una `ufunc` puede devolver múltiples arrays, `modf` es un ejemplo, una versión actualizada del `divmod` incorporado de Python; devuelve una matriz con las partes entera y otra con la parte decimal de cada valor (en punto flotante):

```
In [74]: arr = np.random.randn(7) * 5
         arr
```

```
Out[74]: array([ 0.68, -0.11, -3.1 ,  6.43, -8.89, -0.71,  7.49])
```

```
In [75]: parte_decimal, parte_entera = np.modf(arr)

In [76]: parte_entera

Out[76]: array([ 0., -0., -3.,  6., -8., -0.,  7.])

In [77]: parte_decimal

Out[77]: array([ 0.68, -0.11, -0.1 ,  0.43, -0.89, -0.71,  0.49])
```

Las ufunc aceptan un argumento `out` opcional que les permite operar in situ en los arrays:

```
In [3]: arr = np.array(np.square(np.arange(1,9)), dtype=float)
arr

Out[3]: array([ 1.,  4.,  9., 16., 25., 36., 49., 64.])

In [79]: np.sqrt(arr)

Out[79]: array([1., 2., 3., 4., 5., 6., 7., 8.])

In [80]: arr

Out[80]: array([ 1.,  4.,  9., 16., 25., 36., 49., 64.])

In [4]: np.sqrt(arr, out=arr)
arr

Out[4]: array([1., 2., 3., 4., 5., 6., 7., 8.])
```

Las siguientes tablas muestran las funciones universales disponibles. **Unarias:**

Función	Descripción
abs, fabs	Valor absoluto de los elementos para valores enteros, en coma flotante y complejos
sqrt	Raíz cuadrada de cada elemento, equivalente a <code>arr ** 0.5</code>
square	Cuadrado de cada elemento, equivalente a <code>arr ** 2</code>
exp	Exponencial de cada elemento, $e^x$
log, log10, log2, log1p	Cálculo de logaritmos
sign	Comprueba el signo de cada elemento: 1 (positivo), 0 (cero) y -1 (negativo)
ceil	Calcula el 'techo' de cada elemento (es decir, el menor entero mayor o igual a ese número)
floor	Calcula el 'suelo' de cada elemento (es decir, el mayor entero menor o igual a ese número)
rint	Redondea el elemento al entero más cercano
modf	Devuelve dos matrices con las partes entera y decimal de cada valor
isnan	Booleano indicando si cada valor es NaN ( <i>Not a Number</i> )
isfinite, isinf	Booleano indicando si cada valor es finito o infinito
cos, cosh, sin, sinh, tan, tanh	Funciones trigonométricas regulares e hiperbólicas.
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Funciones trigonométricas inversas
logical_not	Negación del valor, equivalente a <code>~arr</code>

Binarias:

Función	Descripción
add	Suma los elementos correspondientes de los arrays
subtract	Resta los elementos del segundo array a los correspondientes del primer array
multiply	Multiplica los elementos de los arrays
divide, floor_divide	Divide los elementos del array, en el segundo caso truncando el resto
power	Eleva los elementos del primer array a las potencias indicadas por el segundo array
maximum, fmax	valor máximo, <code>fmax</code> ignora NaN
minimum, fmin	valor mínimo, <code>fmin</code> ignora NaN
mod	Resto de la división
copysign	Copia el signo de los elementos del segundo array en los elementos del primer array
greater, greater_equal, less, less_equal, equal, not_equal	Operaciones de comparación
logical_and, logical_or, logical_xor	Operaciones lógicas

**Nota:** En el caso de los operadores de comparación, Numpy los implementa con `ufunc` . Por ejemplo, el código `x < 3` , internamente se ejecuta como `np.less(x, 3)` .

Características avanzadas

Muchas operaciones unarias, como calcular la suma de todos los elementos de la matriz, se implementan como métodos de la clase `ndarray` . De forma predeterminada, estas operaciones se aplican a la matriz como si fuera una lista de números, independientemente de su forma. Sin embargo, al especificar el parámetro de eje ( `axis` ), se puede aplicar una operación a lo largo del eje especificado de una matriz.

```
In [82]: print(A)
A.sum()

[[ 2  3  1]
 [ 1  2  1]
 [-1  4  0]]

Out[82]: 13

In [83]: A.max()

Out[83]: 4

In [84]: A.min()

Out[84]: -1

In [85]: A.sum(axis=0) # suma por columnas

Out[85]: array([2, 9, 2])

In [86]: A.sum(axis=1) # suma por filas

Out[86]: array([6, 4, 3])
```

Especialmente en cálculos grandes, por su ahorro de memoria, es útil poder especificar la matriz donde se almacenará el resultado del cálculo en lugar de crear una matriz temporal. Para ello se puede utilizar el argumento `out` de estas funciones:

```
In [18]: x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(y)

[ 0. 10. 20. 30. 40.]
```

```
In [19]: y = np.zeros(10)
np.power(2, x, out=y[::2])
print(y)

[ 1.  0.  2.  0.  4.  0.  8.  0. 16.  0.]
```

El código equivalente `y[::2] = 2 ** x`, supone la creación de una matriz temporal para contener los resultados de `2 ** x`, seguida de una segunda operación copiando esos valores en la matriz `y`.

En las funciones `ufunc` binarias, hay algunos agregados interesantes que se pueden calcular directamente desde el objeto. Por ejemplo, si queremos reducir una matriz con una operación en particular, podemos usar el método `reduce` de cualquier `ufunc`.

```
In [26]: x = np.arange(1, 6)
np.add.reduce(x)
```

```
Out[26]: 15
```

```
In [27]: np.multiply.reduce(x)
```

```
Out[27]: 120
```

Si queremos almacenar todos los resultados intermedios del cálculo, podemos usar `accumulate` en su lugar:

```
In [28]: np.add.accumulate(x)
```

```
Out[28]: array([ 1,  3,  6, 10, 15])
```

Finalmente, cualquier `ufunc` puede calcular la salida de todos los pares de dos entradas diferentes usando el método `outer`:

```
In [29]: x = np.arange(1, 6)
np.multiply.outer(x, x)
```

```
Out[29]: array([[ 1,  2,  3,  4,  5],
               [ 2,  4,  6,  8, 10],
               [ 3,  6,  9, 12, 15],
               [ 4,  8, 12, 16, 20],
               [ 5, 10, 15, 20, 25]])
```

## Creando nuevas funciones ufunc

Es posible escribir funciones propias que sean aplicables a matrices, mediante el uso de las funciones `frompyfunc` y `vectorize`. La primera siempre devuelve arrays de objetos Python, mientras que la segunda permite especificar el tipo de retorno. Es importante recalcar que estas funciones proporcionan una forma de crear funciones similares a ufunc, pero son muy lentas porque requieren una llamada a la función Python para calcular cada elemento, si se quieren realizar versiones más óptimas habría que utilizar el API C de Numpy o Numba.

```
In [87]: def add_elements(x, y):
          return x + y

          add_all = np.frompyfunc(add_elements, 2, 1)
          add_all(np.arange(8), np.arange(8))
```

```
Out[87]: array([0, 2, 4, 6, 8, 10, 12, 14], dtype=object)
```

```
In [88]: add_them = np.vectorize(add_elements, otypes=[np.float64])
          add_them(np.arange(8), np.arange(8))
```

```
Out[88]: array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14.])
```

## Agregar y eliminar elementos

Existen funciones similares a las vista en Python para trabajar con arrays como `append`, `insert` y `delete` que permiten anadir y eliminar elementos de los arrays. Todas admiten el parámetro opcional `axis` que permite indicar el eje a lo largo del cual se agregan los valores. Si no se indica el eje, tanto la matriz como los valores se aplanan antes de realizar la operación. En el caso de `append` se agregan valores al final de una matriz, los valores se agregan a una copia de la matriz (se asigna y se crea una nueva matriz).

```
In [89]: # aplana las matrices, crea una nueva matriz
          np.append([1, 2, 3], [[4, 5, 6], [7, 8, 9]])
```

```
Out[89]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [90]: # crea una nueva matriz
          np.append([[1, 2, 3], [4, 5, 6]], [[7, 8, 9]], axis=0)
```

```
Out[90]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

```
In [91]: # todas las matrices de entrada deben tener el mismo número de dimensiones
np.append([[1, 2, 3], [4, 5, 6]], [7, 8, 9], axis=0)

-----
ValueError                                Traceback (most recent call last)
<ipython-input-91-185b0c305811> in <module>
      1 # todas las matrices de entrada deben tener el mismo número de dimensi
ones
----> 2 np.append([[1, 2, 3], [4, 5, 6]], [7, 8, 9], axis=0)

~/anaconda3/lib/python3.7/site-packages/numpy/lib/function_base.py in append(a
rr, values, axis)
    4692         values = ravel(values)
    4693         axis = arr.ndim-1
-> 4694     return concatenate((arr, values), axis=axis)
    4695
    4696

ValueError: all the input arrays must have same number of dimensions
```

En el caso de `insert` se insertar valores a lo largo del eje especificado antes del índice o índices dados en su segundo parámetro. Soporta múltiples inserciones cuando este parámetro es un solo escalar o una secuencia con un elemento (similar a llamar a insertar varias veces). Los valores se insertan a una copia de la matriz (se asigna y se crea una nueva matriz).

```
In [92]: a = np.array([[1, 1], [2, 2], [3, 3]])
a
```

```
Out[92]: array([[1, 1],
               [2, 2],
               [3, 3]])
```

```
In [93]: # aplana la matriz, no modifica la matriz original
np.insert(a, 1, 5)
```

```
Out[93]: array([1, 5, 1, 2, 2, 3, 3])
```

```
In [94]: # no modifica la matriz original
np.insert(a, 1, 5, axis=1)
```

```
Out[94]: array([[1, 5, 1],
               [2, 5, 2],
               [3, 5, 3]])
```

```
In [95]: np.insert(a, 1, [5, 6, 7], axis=1)
```

```
Out[95]: array([[1, 5, 1],
               [2, 6, 2],
               [3, 7, 3]])
```

```
In [96]: # conversión de tipos a la matriz origen
np.insert(a, [2, 2], [7.13, False])
```

```
Out[96]: array([1, 1, 7, 0, 2, 2, 3, 3])
```

En el caso de `delete` devuelve una nueva matriz con sub-matrices a lo largo de un eje eliminado. Para una matriz unidimensional, esto devuelve las entradas no devueltas por `arr[obj]`. A menudo es preferible usar una máscara booleana.

```
In [97]: a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])  
a
```

```
Out[97]: array([[ 1,  2,  3,  4],  
               [ 5,  6,  7,  8],  
               [ 9, 10, 11, 12]])
```

```
In [98]: # elimina la fila 1, es una vista  
np.delete(a, 1, axis=0)
```

```
Out[98]: array([[ 1,  2,  3,  4],  
               [ 9, 10, 11, 12]])
```

```
In [99]: # el array original no ha sido modificado  
a
```

```
Out[99]: array([[ 1,  2,  3,  4],  
               [ 5,  6,  7,  8],  
               [ 9, 10, 11, 12]])
```

```
In [100]: # np.delete(arr, [1,3,5], None) equivalente  
np.delete(a, [1,3,5])
```

```
Out[100]: array([ 1,  3,  5,  7,  8,  9, 10, 11, 12])
```

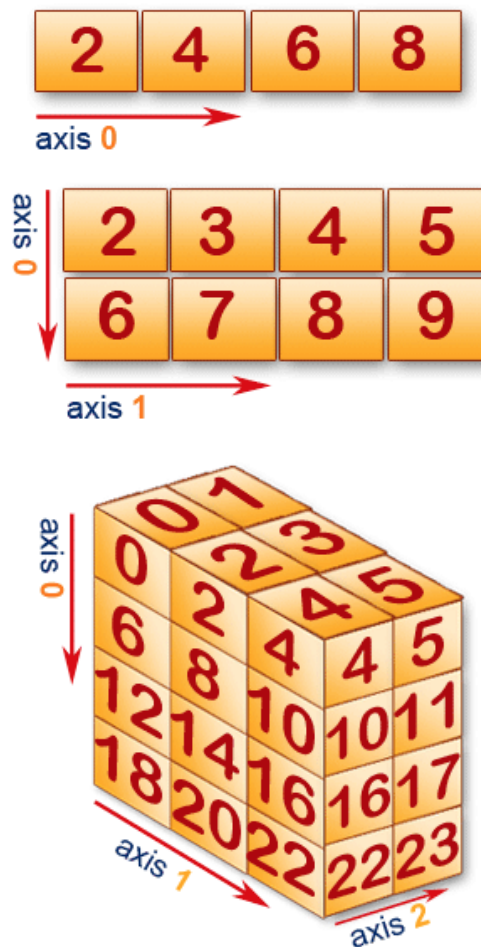
```
In [101]: # elimina las columnas 0 y 2  
np.delete(a, [0,2], 1)
```

```
Out[101]: array([[ 2,  4],  
               [ 6,  8],  
               [10, 12]])
```



## Manipulación de dimensiones

La forma, las dimensiones, de una matriz se puede cambiar con varios comandos. La distribución de los ejes en base a las dimensiones de las matrices es la siguiente:



```
In [102]: A = np.arange(15)
          A
```

```
Out[102]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
In [103]: # 'reshape' devuelve un nuevo array con las dimensiones modificadas
          A.reshape(3,5)
```

```
Out[103]: array([[ 0,  1,  2,  3,  4],
                 [ 5,  6,  7,  8,  9],
                 [10, 11, 12, 13, 14]])
```

```
In [104]: # 'reshape' no modifica el array original
          A
```

```
Out[104]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
In [105]: # Una dimensión puede ser -1. En este caso, el valor se infiere a
# partir de la longitud de la matriz y las dimensiones restantes.
A.reshape(5,-1)
```

```
Out[105]: array([[ 0,  1,  2],
 [ 3,  4,  5],
 [ 6,  7,  8],
 [ 9, 10, 11],
 [12, 13, 14]])
```

```
In [106]: # La nueva dimensión debe ser compatible con la forma original.
# La matriz debe tener el mismo número de elementos.
A.reshape(3,6)
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-106-2572fc6967bb> in <module>
      1 # La nueva dimensión debe ser compatible con la forma original.
      2 # La matriz debe tener el mismo número de elementos.
----> 3 A.reshape(3,6)

ValueError: cannot reshape array of size 15 into shape (3,6)
```

```
In [107]: # También es posible cambiar la forma de una matriz asignando una tupla a la pr
# opiedad 'shape'
# en este caso el cambio es permanente
A.shape = (3, 5)
A
```

```
Out[107]: array([[ 0,  1,  2,  3,  4],
 [ 5,  6,  7,  8,  9],
 [10, 11, 12, 13, 14]])
```

Un patrón de remodelación muy común es la conversión de una matriz unidimensional en una matriz bidimensional de filas o columnas. Aunque se puede hacer con el método `reshape` es más sencillo haciendo uso de la palabra clave `newaxis` dentro de una operación de corte (explicadas a continuación):

```
In [13]: X = np.array([1, 2, 3])
X.reshape((1, 3))
```

```
Out[13]: array([[1, 2, 3]])
```

```
In [14]: X[np.newaxis, :]
```

```
Out[14]: array([[1, 2, 3]])
```

```
In [15]: X.reshape((3, 1))
```

```
Out[15]: array([[1],
 [2],
 [3]])
```

```
In [16]: X[:, np.newaxis]
```

```
Out[16]: array([[1],
 [2],
 [3]])
```

```
In [108]: # El método 'resize' modifica la matriz referenciada.
A.resize(5, 3)
A
```

```
Out[108]: array([[ 0,  1,  2],
                 [ 3,  4,  5],
                 [ 6,  7,  8],
                 [ 9, 10, 11],
                 [12, 13, 14]])
```

```
In [109]: # 'resize' permite aumentar el número de elementos de la matriz si no existen r
eferencias activas
A.resize(3, 6)
A
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-109-396889eb3cee> in <module>
      1 # 'resize' permite aumentar el número de elementos de la matriz si no
    existen referencias activas
----> 2 A.resize(3, 6)
      3 A

ValueError: cannot resize an array that references or is referenced
by another array in this way.
Use the np.resize function or refcheck=False
```

```
In [110]: # utilizando el parámetro 'refcheck=False' se obvian las referencias
A.resize(3, 6, refcheck=False)
A
```

```
Out[110]: array([[ 0,  1,  2,  3,  4,  5],
                 [ 6,  7,  8,  9, 10, 11],
                 [12, 13, 14,  0,  0,  0]])
```

La operación opuesta de remodelar de una dimensión a una dimensión más alta se conoce típicamente como aplanamiento (*flattening*) o desbaste (*raveling*):

```
In [111]: # El método 'ravel' devuelve una nueva matriz plana si los datos originales no
eran contiguos
A.ravel()
```

```
Out[111]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,  0,  0,
                  0])
```

```
In [112]: # El método 'flatten' devuelve siempre una nueva matriz plana
A.flatten()
```

```
Out[112]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,  0,  0,
                  0])
```

### Transposición de matrices e intercambio de ejes

La transposición es una forma especial de remodelación que, de manera similar, devuelve una vista de los datos subyacentes sin copiar nada. Las matrices tienen el método `transpose` y también el atributo especial `T`:

```
In [113]: A.resize(3, 5, refcheck=False)
          A.T # equivalente a A.reshape(5,3)
```

```
Out[113]: array([[ 0,  5, 10],
                 [ 1,  6, 11],
                 [ 2,  7, 12],
                 [ 3,  8, 13],
                 [ 4,  9, 14]])
```

Para matrices de dimensiones superiores, el método `transpose` acepta una tupla de números para permutar los ejes (para una mayor flexibilidad):

```
In [114]: A = np.arange(16).reshape((2, 2, 4))
          A
```

```
Out[114]: array([[[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7]],

                 [[ 8,  9, 10, 11],
                  [12, 13, 14, 15]]])
```

```
In [115]: # los ejes se han reordenado, con el segundo eje primero, el primer eje segundo y el resto igual
          A.transpose((1, 0, 2))
```

```
Out[115]: array([[[ 0,  1,  2,  3],
                  [ 8,  9, 10, 11]],

                 [[ 4,  5,  6,  7],
                  [12, 13, 14, 15]]])
```

La transposición simple con `T` es un caso especial de cambio de ejes. Los `ndarray` tienen el método `swapaxes`, que toma un par de números de eje y cambia los ejes indicados para reordenar los datos:

```
In [116]: A
```

```
Out[116]: array([[[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7]],

                 [[ 8,  9, 10, 11],
                  [12, 13, 14, 15]]])
```

```
In [117]: A.swapaxes(1, 2)
```

```
Out[117]: array([[[ 0,  4],
                  [ 1,  5],
                  [ 2,  6],
                  [ 3,  7]],

                 [[ 8, 12],
                  [ 9, 13],
                  [10, 14],
                  [11, 15]])
```

## Indexación y Partición básicas

Las matrices unidimensionales se pueden indexar, dividir e iterar, como las listas y otras secuencias de Python.

```
In [118]: arr = np.arange(10)
arr
```

```
Out[118]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [119]: arr[5]
```

```
Out[119]: 5
```

```
In [120]: arr[5:8]
```

```
Out[120]: array([5, 6, 7])
```

```
In [121]: # asignar un valor escalar a una partición hace que el valor se copie en todos
los elementos del segmento
arr[5:8] = 12
arr
```

```
Out[121]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

Una primera distinción importante respecto de las listas integradas de Python es que los segmentos de una matriz son *vistas* en la matriz original. Esto significa que los datos no se copian, y cualquier modificación a la *vista* se reflejará en la matriz de origen.

```
In [122]: # las secciones son vistas
arr_slice = arr[5:8]
arr_slice
```

```
Out[122]: array([12, 12, 12])
```

```
In [123]: arr_slice[:] = 64
arr
```

```
Out[123]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

Si desea una copia de un segmento de un `ndarray` en lugar de una vista, se deberá utilizar el método `copy` :

```
In [124]: arr_slice_copy = arr[5:8].copy()
```

Con matrices de dimensiones superiores, hay muchas más opciones. En una matriz bidimensional, los elementos de cada índice ya no son escalares sino matrices unidimensionales. Cuando se proporcionan menos índices que el número de ejes, los índices faltantes se consideran segmentos completos:

```
In [125]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2d
```

```
Out[125]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

```
In [126]: arr2d[-1] # es equivalente a arr2d[2]
```

```
Out[126]: array([7, 8, 9])
```

Así, se puede acceder recursivamente a elementos individuales. También se puede pasar una lista de índices separados por comas para seleccionar elementos individuales:

```
In [127]: arr2d[0][2]
```

```
Out[127]: 3
```

```
In [128]: arr2d[0, 2]
```

```
Out[128]: 3
```

La siguiente imagen muestra la indexación en una matriz bidimensional. Es útil pensar en el eje 0 como las "filas" de la matriz y el eje 1 como las "columnas":




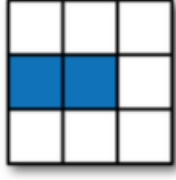
		axis 1		
		0	1	2
axis 0	0	0,0	0,1	0,2
	1	1,0	1,1	1,2
	2	2,0	2,1	2,2

La expresión entre paréntesis en `arr2d[i]` se trata como una `i` seguida de tantas instancias de `:` como sea necesario para representar los ejes restantes. NumPy también te permite escribir esto usando puntos como `arr2d[i, ...]`.

Los puntos (...) representan tantos `:` como sean necesarios para producir una tupla de indexación completa. Por ejemplo, si `x` es una matriz con 5 ejes, entonces

```
x [1,2, ...] equivalente a x [1,2,::,:,:],
x [..., 3] equivalente a a x[:,::,:, :, 3] y
x [4, ..., 5, :] equivalente a x [4,::,:, 5, :].
```

En matrices multidimensionales, si se omiten los índices posteriores, el objeto devuelto será un ndarray de dimensión menor que consta de todos los datos a lo largo de las dimensiones superiores.

	Expression	Shape
	<code>arr[:2, 1:]</code>	<code>(2, 2)</code>
	<code>arr[2]</code> <code>arr[2, :]</code> <code>arr[2:, :]</code>	<code>(3,)</code> <code>(3,)</code> <code>(1, 3)</code>
	<code>arr[:, :2]</code>	<code>(3, 2)</code>
	<code>arr[1, :2]</code> <code>arr[1:2, :2]</code>	<code>(2,)</code> <code>(1, 2)</code>

```
In [129]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]]) # matriz
2 x 2 x 3
arr3d
```

```
Out[129]: array([[[ 1,  2,  3],
                  [ 4,  5,  6]],

                [[ 7,  8,  9],
                 [10, 11, 12]]])
```

```
In [130]: arr3d.shape
```

```
Out[130]: (2, 2, 3)
```

```
In [131]: arr3d[0] # arr3d[0,:,:] array de 2 x 3
```

```
Out[131]: array([[1, 2, 3],
                 [4, 5, 6]])
```

```
In [132]: arr3d[... ,2] # arr3d[:, :, 2] array de 2 x 2
```

```
Out[132]: array([[ 3,  6],
                 [ 9, 12]])
```

```
In [133]: # A un segmento de un array se le pueden asignar valores escalares y arrays
arr3d[0] = 17
arr3d
```

```
Out[133]: array([[[17, 17, 17],
                  [17, 17, 17]],

                [[ 7,  8,  9],
                 [10, 11, 12]]])
```

Otra operativa básica sobre matrices haciendo uso de su capacidad de indexación es invertir sus filas y columnas:

```
In [134]: arr1 = np.arange(1, 13).reshape(3, 4)
arr1
```

```
Out[134]: array([[ 1,  2,  3,  4],
                 [ 5,  6,  7,  8],
                 [ 9, 10, 11, 12]])
```

```
In [135]: # invertir la posición de las filas
arr1[::-1, ]
```

```
Out[135]: array([[ 9, 10, 11, 12],
                 [ 5,  6,  7,  8],
                 [ 1,  2,  3,  4]])
```

```
In [136]: # invertir la posición de las filas y las columnas
arr1[::-1, ::-1]
```

```
Out[136]: array([[12, 11, 10,  9],
                 [ 8,  7,  6,  5],
                 [ 4,  3,  2,  1]])
```

## Indexación avanzada

NumPy ofrece más facilidades de indexación que las secuencias normales de Python. Además de la indexación por enteros y segmentos, como vimos antes, las matrices se pueden indexar por matrices de enteros y matrices de valores booleanos.

### Indexación con matrices de índices (Fancy Indexing)

```
In [137]: a = np.arange(12)**2 # los 12 primeros cuadrados
a
```

```
Out[137]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81, 100, 121])
```

```
In [138]: a[[1, 3, 5, 7]] # los elementos de a indicados en el índice
```

```
Out[138]: array([ 1,  9, 25, 49])
```

```
In [139]: a[np.array([[3, 4], [9, 7]])] # los elementos indicados con la forma de los índices
```

```
Out[139]: array([[ 9, 16],
                 [81, 49]])
```

Cuando la matriz indexada es multidimensional, una matriz única de índices se refiere a la primera dimensión de la matrix indexada.



```
In [140]: paleta = np.array( [
    [0,0,0],           # negro
    [255,0,0],         # rojo
    [0,255,0],         # verde
    [0,0,255],         # azul
    [255,255,255] ] )
paleta
```

```
Out[140]: array([[ 0,  0,  0],
 [255,  0,  0],
 [  0, 255,  0],
 [  0,  0, 255],
 [255, 255, 255]])
```

```
In [141]: # cada valor se corresponde con un color de la paleta
imagen = np.array( [ [ 0, 1, 2, 0 ], [ 0, 3, 4, 0 ] ] )
imagen
```

```
Out[141]: array([[0, 1, 2, 0],
 [0, 3, 4, 0]])
```

```
In [142]: paleta[imagen]
```

```
Out[142]: array([[[ 0,  0,  0],
 [255,  0,  0],
 [  0, 255,  0],
 [  0,  0,  0]],

 [[ 0,  0,  0],
 [  0,  0, 255],
 [255, 255, 255],
 [  0,  0,  0]])
```

También podemos dar índices para más de una dimensión. Las matrices de índices para cada dimensión deben tener la misma forma.

```
In [143]: a = np.arange(12).reshape(3,4)
a
```

```
Out[143]: array([[ 0,  1,  2,  3],
 [ 4,  5,  6,  7],
 [ 8,  9, 10, 11]])
```

```
In [144]: i = np.array( [ [0,1], [1,2] ] ) # indices para la primera dimensión de a
j = np.array( [ [2,1], [3,3] ] ) # indices para la segunda dimensión de a
```

```
In [145]: a[i] # equivalente a a[i,:]
```

```
Out[145]: array([[[ 0,  1,  2,  3],
 [ 4,  5,  6,  7]],

 [[ 4,  5,  6,  7],
 [ 8,  9, 10, 11]])
```

Define una matriz de 3 dimensiones:

```
array([[[ 0,  1,  2,  3],      # fila 0 -> a[i][0][0]
 [ 4,  5,  6,  7]],         # fila 1 -> a[i][0][1]

 [[ 4,  5,  6,  7],         # fila 1 -> a[i][1][0]
 [ 8,  9, 10, 11]])        # fila 2 -> a[i][1][1]
```

```
In [146]: a[i,j] # i y j deben tener la misma forma
```

```
Out[146]: array([[ 2,  5],
                [ 7, 11]])
```

Se seleccionan de las filas de `a[i]` las columnas indicadas por `j` :

```
array([[ 0,  1, <2>,  3],      # columna 2
       [ 4, <5>,  6,  7]],     # columna 1

      [[ 4,  5,  6, <7>],      # columna 3
       [ 8,  9, 10, <11>]])    # columna 3
```

También se puede utilizar la indexación con matrices como destino para asignaciones:

```
In [147]: a = np.arange(5)
a
```

```
Out[147]: array([0, 1, 2, 3, 4])
```

```
In [148]: a[[1,3,4]] = 0
a
```

```
Out[148]: array([0, 0, 2, 0, 0])
```

Sin embargo, cuando la lista de índices contiene repeticiones, la asignación se realiza varias veces, dejando siempre el último valor:

```
In [149]: a = np.arange(5)
a[[0,0,2]] = [1,2,3]
a
```

```
Out[149]: array([2, 1, 3, 3, 4])
```

Existen métodos alternativos de `ndarray` que son útiles en el caso especial de solo hacer una selección en un solo eje, los métodos `take` y `put` :

```
In [150]: arr = np.arange(10) * 100
inds=[7,1,2,6]
arr.take(inds) # equivalente a arr[[7,1,2,6]] o arr[inds]
```

```
Out[150]: array([700, 100, 200, 600])
```

```
In [151]: arr.put(inds, 42)
arr
```

```
Out[151]: array([  0,  42,  42, 300, 400, 500,  42,  42, 800, 900])
```

```
In [152]: arr.put(inds, [40, 41, 42, 43])
arr
```

```
Out[152]: array([  0,  41,  42, 300, 400, 500,  43,  40, 800, 900])
```

El método `take` permite la selección de índices en diferentes ejes mediante el uso de el parámetro `axis` , sin embargo `put` no acepta un argumento de eje:

```
In [153]: inds = [2, 0, 2, 1]
arr = np.random.randint(8, size=(2, 4))
arr
```

```
Out[153]: array([[5, 1, 0, 3],
                [0, 2, 5, 1]])
```

```
In [154]: arr.take(inds, axis=1)
```

```
Out[154]: array([[0, 5, 0, 1],
                [5, 0, 5, 2]])
```

## Indexación booleana

Cuando indexamos matrices con matrices de índices (enteros), proporcionamos la lista de índices para seleccionar. Con los índices booleanos el enfoque es diferente; elegimos explícitamente los elementos de la matriz que queremos y los que no.

La forma más natural en que se puede pensar para la indexación booleana es utilizar matrices booleanas que tengan la misma forma que la matriz original:

```
In [155]: a = np.arange(12).reshape(3,4)
b = a > 4
b # b es una matriz booleana con la forma de a
```

```
Out[155]: array([[False, False, False, False],
                [False,  True,  True,  True],
                [ True,  True,  True,  True]])
```

```
In [156]: a[b] # array de 1 dimensión con los elementos seleccionados
```

```
Out[156]: array([ 5,  6,  7,  8,  9, 10, 11])
```

Esta propiedad puede ser muy útil en asignaciones:

```
In [157]: a[b] = 0 # todos los elementos de a mayores que 4 se inicializan a 0
a
```

```
Out[157]: array([[0, 1, 2, 3],
                [4, 0, 0, 0],
                [0, 0, 0, 0]])
```

```
In [158]: a[~b] = 1 # todos los elementos de a menores o igual que 4 se inicializan a 1
a
```

```
Out[158]: array([[1, 1, 1, 1],
                [1, 0, 0, 0],
                [0, 0, 0, 0]])
```

La segunda forma de indexar con booleanos es más similar a la indexación de enteros. Para cada dimensión de la matriz, proporcionamos una matriz booleana 1D que selecciona los segmentos que deseamos:

```
In [159]: a = np.arange(12).reshape(3,4)
a
```

```
Out[159]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])
```

```
In [160]: b1 = np.array([False,True,True])           # selección de la primera dimensio
n
          b2 = np.array([True,False,True,False])    # selección de la segunda dimensio
n
```

```
In [161]: a[b1]  # seleccionamos filas, equivalente a a[b1,:]
```

```
Out[161]: array([[ 4,  5,  6,  7],
                 [ 8,  9, 10, 11]])
```

```
In [162]: a[:, b2]  # seleccionamos columnas
```

```
Out[162]: array([[ 0,  2],
                 [ 4,  6],
                 [ 8, 10]])
```

```
In [163]: a[b1, b2]  # seleccionamos filas y columnas
```

```
Out[163]: array([ 4, 10])
```

## Concatenación y división de arrays

La función `np.concatenate` toma una secuencia (tupla, lista, etc.) de matrices y las une en orden a lo largo del eje de entrada indicado:

```
In [164]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])
          arr2 = np.array([[7, 8, 9], [10, 11, 12]])
          np.concatenate([arr1, arr2], axis=0)
```

```
Out[164]: array([[ 1,  2,  3],
                 [ 4,  5,  6],
                 [ 7,  8,  9],
                 [10, 11, 12]])
```

```
In [165]: np.concatenate([arr1, arr2], axis=1)
```

```
Out[165]: array([[ 1,  2,  3,  7,  8,  9],
                 [ 4,  5,  6, 10, 11, 12]])
```

Existen funciones específicas como `vstack` y `hstack`, para estos tipos comunes de concatenación. Las operaciones anteriores podrían haberse expresado como:

```
In [166]: np.vstack((arr1, arr2))
```

```
Out[166]: array([[ 1,  2,  3],
                 [ 4,  5,  6],
                 [ 7,  8,  9],
                 [10, 11, 12]])
```

```
In [167]: np.hstack((arr1, arr2))
```

```
Out[167]: array([[ 1,  2,  3,  7,  8,  9],
                 [ 4,  5,  6, 10, 11, 12]])
```

Por otro lado, la función `split` divide una matriz en múltiples matrices a lo largo de un eje:

```
In [168]: arr1 = np.concatenate([arr1, arr2], axis=0)
          first, second, third = np.split(arr1, [1, 3])
          # El valor [1, 3] pasado a np.split indica los índices en los que se divide la
          matriz en partes
          first
```

```
Out[168]: array([[1, 2, 3]])
```

```
In [169]: second
```

```
Out[169]: array([[4, 5, 6],
                 [7, 8, 9]])
```

```
In [170]: third
```

```
Out[170]: array([[10, 11, 12]])
```

La siguiente tabla muestra las funciones de concatenación/división de arrays:

Función	Descripción
<code>concatenate</code>	Concatena un conjunto de arrays a lo largo de un eje
<code>vstack</code> , <code>row_stack</code>	Concatena matrices por filas (a lo largo del eje 0)
<code>hstack</code>	Concatena matrices por columnas (a lo largo del eje 1)
<code>column_stack</code>	Igual que <code>hstack</code> , pero primero convierte matrices 1D a vectores 2D
<code>dstack</code>	Concatena matrices en "profundidad" (a lo largo del eje 2)
<code>split</code>	Dividir una matriz en múltiples matrices a lo largo de un eje
<code>hsplit</code> , <code>vsplit</code>	Dividir una matriz en múltiples matrices a lo largo de un eje 0 y 1 respectivamente

## Recorrido sobre NumPy Arrays

La iteración básica sobre matrices multidimensionales se realiza con respecto al primer eje:

```
In [171]: arr2d = np.arange(12).reshape(3,4)
          for fila in arr2d:
              print(fila)

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

Si se quiere realizar una operación en cada elemento de la matriz podríamos hacer 2 bucles anidados.

```
In [172]: for fila in arr2d:
          for columna in fila:
              print(columna, end=" ")

0 1 2 3 4 5 6 7 8 9 10 11
```

Sin embargo, es más óptimo utilizar el atributo `flat` que proporciona un iterador sobre todos los elementos de la matriz.

```
In [173]: for elemento in arr2d.flat:
           print(elemento, end=" ")
```

```
0 1 2 3 4 5 6 7 8 9 10 11
```

Adicionalmente a los métodos de recorridos de listas/arrays vistos en Python, Numpy incluye el iterador `nditer` que proporciona formas flexibles de recorrer todos los elementos de uno o más arrays de una manera sistemática. `nditer` proporciona un mapeo relativamente sencillo de la API del iterador de arrays en C, que mejora los tiempos de ejecución.

### Iteración sobre una única matriz

La tarea más básica que se puede hacer con el `nditer` es iterar cada elemento de una matriz. Cada elemento se proporciona **uno por uno** utilizando la interfaz de iterador de Python estándar.

```
In [174]: arr2d = np.arange(6).reshape(2,3)
           for x in np.nditer(arr2d):
               print(x, end=' ')
```

```
arr2d
```

```
0 1 2 3 4 5
```

```
Out[174]: array([[0, 1, 2],
                 [3, 4, 5]])
```

### Iteración sobre una varias matrices

Una de las funcionalidades básicas de `nditer` es iterar de forma coordinada por conjuntos de matrices, en una funcionalidad similar al uso de la función `zip` en Python.

```
In [175]: for x, y in np.nditer([arr2d, arr2d**2]):
           print("%d^2=%d" % (x,y), end=' ')
```

```
0^2=0    1^2=1    2^2=4    3^2=9    4^2=16    5^2=25
```

### Orden en memoria y orden de en recorrido

Por razones históricas, el orden por fila y el orden por columna también se conocen como orden C y orden Fortran, respectivamente. El orden de los elementos una matriz es normalmente "estilo C", pero los métodos/funciones que trabajan con matrices suelen admitir un parámetro `order` que permite cambiar este comportamiento.

La siguiente imagen muestra con el parámetro `order` afecta a las operaciones de manipulación de dimensiones:

0	1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	---	----	----

`arr.reshape((4, 3), order=?)`

**C order (row major)**

0	1	2
3	4	5
6	7	8
9	10	11

`order='C'`

**Fortran order (column major)**

0	4	8
1	5	9
2	6	10
3	7	11

`order='F'`

Una cosa importante a tener en cuenta cuando iteramos es que el orden se elige para que coincida con el diseño de la memoria de la matriz. Esto se hace para mejorar la eficiencia de acceso, reflejando la idea de que, de forma predeterminada, uno simplemente quiere visitar cada elemento sin preocuparse por un orden en particular.

```
In [176]: # el recorrido no se aplica a la vista sino al diseño de memoria
print(arr2d.T, end=" -> ")
for x in np.nditer(arr2d.T):
    print(x, end=' ')
```

```
[[0 3]
 [1 4]
 [2 5]] -> 0 1 2 3 4 5
```

```
In [177]: # el recorrido no se aplica a la vista sino al diseño de memoria
arr2dF = arr2d.copy(order='F')
print(arr2dF, end=" -> ")
for x in np.nditer(arr2dF):
    print(x, end=' ')
```

```
[[0 1 2]
 [3 4 5]] -> 0 3 1 4 2 5
```

### Control del orden de iteración

Hay ocasiones en que es importante visitar los elementos de una matriz en un orden específico, independientemente del diseño de los elementos en la memoria. El objeto `nditer` proporciona un parámetro de orden para controlar este aspecto de la iteración. El valor predeterminado es `order = 'K'` que mantiene la disposición de memoria y que se puede anular forzando los valores a `'C'` para orden C o `'F'` para orden Fortran.

```
In [178]: for x in np.nditer(arr2dF, order='C'):
           print(x, end=' ')

0 1 2 3 4 5
```

### Modificación de valores de matriz

De forma predeterminada, `nditer` trata el operando de entrada como un objeto de solo lectura. Para poder modificar los elementos de la matriz, se debe especificar el modo de lectura/escritura o solo de escritura utilizando los indicadores `readwrite` o `writeonly`.

El `nditer` producirá matrices modificables. Sin embargo, debido a que `nditer` debe copiar los datos del búfer de nuevo a la matriz original una vez finalizada la iteración, se debe indicar cuándo finaliza la iteración, mediante uno de los dos métodos:

- Usando `nditer` en el contexto de una instrucción `with`, y los datos temporales se escribirán cuando se salga del contexto.
- Llamando al método de cierre del iterador `close` una vez que termine de iterar, lo que activará la reescritura.

El iterador creado con `nditer` no se podrá utilizar una vez que se llama a `close` o se sale de su contexto:

```
In [179]: with np.nditer(arr2d, op_flags=['readwrite']) as it:
           for x in it:
               x[...] = x * x
           arr2d

Out[179]: array([[ 0,  1,  4],
                 [ 9, 16, 25]])
```

### Usando un bucle externo

En todos los ejemplos anteriores los elementos del array son proporcionados por el iterador uno cada vez, porque toda la lógica de bucle es interna al iterador. Si bien esto es simple y conveniente, no es muy eficiente. Un enfoque mejor es mover el bucle unidimensional más interno del código, al exterior del iterador. De esta manera, las operaciones vectorizadas de NumPy se pueden usar en porciones más grandes de los elementos que se visitan.

El `nditer` intentará proporcionar trozos que sean lo más grandes posible para el bucle interno. Al forzar el orden de "C" y "F", obtenemos diferentes tamaños de bucle externo. Este modo se habilita especificando un flag del iterador.

Observe que con el valor predeterminado de mantener el orden de la memoria nativa, el iterador puede proporcionar un solo fragmento unidimensional, mientras que al forzar el orden de Fortran, tiene que proporcionar tres fragmentos de dos elementos cada uno:

```
In [180]: for x in np.nditer(arr2d, flags=['external_loop']):
           print(x, end=' ')

[ 0  1  4  9 16 25]
```

```
In [181]: for x in np.nditer(arr2d, flags=['external_loop'], order='F'):
           print(x, end=' ')

[0 9] [ 1 16] [ 4 25]
```



### Almacenando los elementos del array

Al forzar un orden de iteración, observamos que la opción de bucle externo puede proporcionar los elementos en partes más pequeñas porque los elementos no se pueden visitar en el orden apropiado con un paso constante. Al escribir código C, esto generalmente es correcto, sin embargo, en el código puro de Python, esto puede causar una reducción significativa en el rendimiento.

Al habilitar el modo de almacenamiento en búfer, los fragmentos proporcionados por el iterador al bucle interno pueden hacerse más grandes, reduciendo significativamente la sobrecarga del intérprete de Python. En el ejemplo de forzar el orden de iteración de Fortran, el bucle externo puede ver todos los elementos de una sola vez cuando el búfer está habilitado.

```
In [182]: for x in np.nditer(arr2d, flags=['external_loop', 'buffered'], order='F'):
           print(x, end=' ')

[ 0  9  1 16  4 25]
```

### Seguimiento de un índice o multi-índice

Durante la iteración, es posible que desee utilizar el índice del elemento actual en un cálculo. Por ejemplo, es posible que desee visitar los elementos de una matriz en un cierto orden para buscar valores en una matriz diferente.

El protocolo del iterador de Python no tiene una forma natural de consultar estos valores adicionales del iterador, pero si es posible al iterar con un `nditer`. Esta sintaxis funciona explícitamente con el objeto iterador en sí, por lo que sus propiedades son fácilmente accesibles durante la iteración, utilizando las propiedades `index` o `multi_index` según nuestras necesidades.

El intérprete interactivo de Python desafortunadamente imprime los valores de las expresiones dentro del bucle `while` durante cada iteración del bucle. Se modificó la salida en los ejemplos utilizando esta construcción de bucle para que sea más legible.

```
In [183]: # índice Fortran
it = np.nditer(arr, flags=['f_index'])
while not it.finished:
    print("%d <%d>" % (it[0], it.index), end=' ')
    it.iternext()

5 <0> 1 <2> 0 <4> 3 <6> 0 <1> 2 <3> 5 <5> 1 <7>
```

```
In [184]: it = np.nditer(arr, flags=['multi_index'])
while not it.finished:
    print("%d <%s>" % (it[0], it.multi_index), end=' ')
    it.iternext()

5 <(0, 0)> 1 <(0, 1)> 0 <(0, 2)> 3 <(0, 3)> 0 <(1, 0)> 2 <(1, 1)> 5 <(1, 2)> 1
<(1, 3)>
```

```
In [185]: it = np.nditer(arr, flags=['multi_index'], op_flags=['writeonly'])
with it:
    while not it.finished:
        it[0] = it.multi_index[1] - it.multi_index[0]
        it.iternext()

arr
```

```
Out[185]: array([[ 0,  1,  2,  3],
                 [-1,  0,  1,  2]])
```

**Nota:** El seguimiento de un índice o multi-índice es incompatible con el uso de un bucle externo, ya que requiere un valor de índice diferente por elemento.

### Iterando como un tipo de datos específico

Hay ocasiones en que es necesario tratar los elementos de una matriz como un tipo de datos diferente del que se creó. Excepto cuando se escribe código C de bajo nivel, generalmente es mejor dejar que el iterador maneje una copia de los datos en lugar de modificar el tipo de los datos en el bucle interno.

Hay dos mecanismos que permiten hacer esto, las copias temporales y el modo de almacenamiento en búfer. Con las copias temporales, se realiza una copia de toda la matriz con el nuevo tipo de datos, luego se realiza la iteración en la copia. Se permite el acceso de escritura a través de un modo que actualiza la matriz original después de que se completa toda la iteración. El principal inconveniente de las copias temporales es que la copia temporal puede consumir una gran cantidad de memoria, especialmente si el tipo de datos de iteración tiene un tamaño de elemento mayor que el original.

El modo de almacenamiento en búfer mitiga el problema de uso de la memoria y es más fácil almacenar en caché que hacer copias temporales. Excepto en casos especiales, donde se necesita la matriz completa al mismo tiempo fuera del iterador, se recomienda el almacenamiento en búfer sobre la copia temporal. En NumPy, las `ufuncs` y otras funciones utilizan el almacenamiento en búfer para admitir entradas flexibles con una sobrecarga de memoria mínima.

En nuestros ejemplos, trataremos la matriz de entrada con un tipo de datos complejo, de modo que podamos tomar raíces cuadradas de números negativos. Sin habilitar las copias o el modo de almacenamiento en búfer, el iterador generará una excepción si el tipo de datos no coincide exactamente.

```
In [186]: # los números enteros negativos no tienen raíz cuadrada
arr = np.arange(6).reshape(2,3) - 3
for x in np.nditer(arr):
    print(np.sqrt(x), end=' ')

nan nan nan 0.0 1.0 1.4142135623730951

/Users/abraham/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:4:
RuntimeWarning: invalid value encountered in sqrt
  after removing the cwd from sys.path.
```

```
In [187]: # los números complejos negativos si tienen raíz cuadrada
for x in np.nditer(arr, flags=['buffered'], op_dtypes=['complex']):
    print(np.sqrt(x), end=' ')

1.7320508075688772j 1.4142135623730951j 1j 0j (1+0j) (1.4142135623730951+0j)
```

El iterador utiliza las reglas de conversión de NumPy para determinar si se permite una conversión específica. Por defecto, impone la conversión "segura". Esto significa, por ejemplo, que generará una excepción si se intenta tratar una matriz con float-64 como una matriz float-32 bits. En muchos casos, la regla `same_kind` es la más razonable de usar, ya que permitirá la conversión de float 64 a 32 bits, pero no de float a int o de complejo a float.

```
In [188]: arr = np.arange(6.)
          for x in np.nditer(arr, flags=['buffered'], op_dtypes=['float32']):
              print(x, end=' ')

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-188-203e36fd3ced> in <module>
      1 arr = np.arange(6.)
----> 2 for x in np.nditer(arr, flags=['buffered'], op_dtypes=['float32']):
      3     print(x, end=' ')

TypeError: Iterator operand 0 dtype could not be cast from dtype('float64') to
dtype('float32') according to the rule 'safe'
```

```
In [189]: for x in np.nditer(arr, flags=['buffered'], op_dtypes=['float32'], casting='sam
          e_kind'):
              print(x, end=' ')

0.0 1.0 2.0 3.0 4.0 5.0
```

### Iteradores y difusión en matrices

En el objeto `nditer` también se aplican las reglas de broadcasting/difusión de matrices cuando es necesario. A modo de ejemplo, imprimimos el resultado de un recorrido combinado de dos matrices de diferentes dimensiones:

```
In [190]: arr1 = np.arange(3) # matriz 1D
          arr2 = np.arange(6).reshape(2,3) # matriz 2D
          print(arr1, " shape -> ", arr1.shape, "\n")
          print(arr2, " shape -> ", arr2.shape, "\n")

          for x, y in np.nditer([arr1, arr2]):
              print("%d:%d" % (x,y), end=' ')

[0 1 2] shape -> (3,)

[[0 1 2]
 [3 4 5]] shape -> (2, 3)

0:0 1:1 2:2 0:3 1:4 2:5
```

Las operaciones de difusión se realizan siempre que sea posible adaptar los operandos y uno o ambos se puedan expandir en una o más dimensiones para hacerlos compatibles:

```
In [191]: print(arr1, " shape -> ", arr1.shape)
print(arr2.T, " shape -> ", arr2.T.shape, "\n")
for x, y in np.nditer([arr1, arr2.T]):
    print("%d:%d" % (x,y), end=' ')
```

```
[0 1 2] shape -> (3,)
[[0 3]
 [1 4]
 [2 5]] shape -> (3, 2)
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-191-1d2db390fe63> in <module>
      1 print(arr1, " shape -> ", arr1.shape)
      2 print(arr2.T, " shape -> ", arr2.T.shape, "\n")
----> 3 for x, y in np.nditer([arr1, arr2.T]):
      4     print("%d:%d" % (x,y), end=' ')
```

```
ValueError: operands could not be broadcast together with shapes (3,) (3,2)
```

### Arrays de salida vinculados al iterador

Un caso común en las funciones NumPy es tener salidas asignadas en función del broadcasting de la entrada, y además tener un parámetro opcional llamado `out` que devuelve el resultado. El objeto `nditer` proporciona una sintaxis similar que proporciona este mecanismo:

```
In [192]: def square(a):
          with np.nditer([a, None]) as it:
              for x, y in it:
                  y[...] = x*x
              return it.operands[1]

square([1,2,3])
```

```
Out[192]: array([1, 4, 9])
```

Cuando se pasa un valor `None` a `nditer` de forma predeterminada se fijan los flags `allocate` and `writenonly` para ese operando. Si queremos utilizar un parámetro opcional para indicar un array de entrada, tenemos que proporcionar explícitamente los flags, porque el iterador usará de forma predeterminada `readonly` para evitar problemas. También se debe utilizar el flag `no_broadcast`, que evitará que se emita la salida. Esto es importante, si solo queremos un valor de entrada para cada salida. La agregación de más de un valor de entrada es una operación de reducción que requiere un manejo especial. Para completar, también se debe añadir los distintivos `external_loop` y `buffered`, ya que estos son los que normalmente se utilizan por razones de rendimiento.

```
In [193]: def square(a, out=None):
          it = np.nditer([a, out],
                        flags = ['external_loop', 'buffered'],
                        op_flags = [['readonly'],
                                   ['writenonly', 'allocate', 'no_broadcast']])

          with it:
              for x, y in it:
                  y[...] = x*x
              return it.operands[1]

square([1,2,3])
```

```
Out[193]: array([1, 4, 9])
```

```
In [194]: arr = np.zeros(3)
          square([1,2,3], out=arr)
          arr
```

```
Out[194]: array([1., 4., 9.])
```

## Iteraciones con Reducción

Cuando un operando de escritura tiene menos elementos que el espacio de iteración completo, ese operando está experimentando una reducción. El objeto `nditer` requiere que cualquier operando de reducción se marque como `readwrite`, y solo permite reducciones cuando se proporciona `reduce_ok` como un flag del iterador:

```
In [195]: def nditer_sum(a, b):
          it = np.nditer([a, b],
                        flags = ['reduce_ok'],
                        op_flags = [['readonly'], ['readwrite']])
          with it:
              for x, y in it:
                  y[...] += x
              return int(it.operands[1])

          arr1 = np.arange(24).reshape(2,3,4)
          arr2 = np.array(0)
          nditer_sum(arr1, arr2) # equivalente a np.sum(arr1)
```

```
Out[195]: 276
```

## Programación orientada a Arrays

El uso de matrices NumPy permite definir muchos tipos de tareas de procesamiento de datos como expresiones concisas sobre matrices que de otro modo podrían requerir la escritura de bucles. Esta práctica de reemplazar los bucles explícitos con expresiones matriciales se conoce comúnmente como *vectorización*. En general, las operaciones de matrices vectorizadas a menudo serán uno o dos (o más) órdenes de magnitud más rápidas que sus equivalentes puros de Python, con el mayor impacto en cualquier tipo de cálculo numérico.

Como ejemplo simple, supongamos que deseamos evaluar la función  $\sqrt{(x^2 + y^2)}$  a través de una matriz de valores regular.

La función `np.meshgrid` toma dos matrices 1D y produce dos matrices 2D, de iguales dimensiones, correspondientes a todos los pares de  $(x, y)$  en las dos matrices:

```
In [196]: points = np.arange(-5, 5, 0.01) # 1000 puntos
          xs, ys = np.meshgrid(points, points)
          xs.shape
```

```
Out[196]: (1000, 1000)
```

```
In [197]: import math
          %time z = [math.sqrt(x ** 2 + y ** 2) for x, y in zip(xs.flatten(), ys.flatten())]
          len(z)
```

```
CPU times: user 1.36 s, sys: 81 ms, total: 1.44 s
Wall time: 1.03 s
```

```
Out[197]: 1000000
```

```
In [198]: %time z = np.sqrt(xs ** 2 + ys ** 2)
          z.size
```

CPU times: user 44.4 ms, sys: 13.6 ms, total: 58 ms  
Wall time: 26.9 ms

```
Out[198]: 1000000
```

```
In [199]: import matplotlib.pyplot as plt
          plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
          plt.title("Gráfico de  $\sqrt{x^2 + y^2}$  para una matriz de valores")
          plt.draw();
          # plt.close('all')
```

## Expresando lógica condicional como operaciones sobre Arrays

La función `numpy.where` es una versión vectorizada de la expresión ternaria `x if condición else y`. Supongamos que tenemos una matriz booleana y dos matrices de valores:

```
In [200]: # xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
          # yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
          # cond = np.array([True, False, True, True, False])

          xarr = np.random.randn(10000000)
          yarr = np.random.randn(10000000)
          cond = np.random.choice(a=[False, True], size=10000000)
```

Y queremos tomar un valor de `xarr` siempre que el valor correspondiente en `cond` sea `True`, y de lo contrario tomar el valor de `yarr`. Con una lista de comprensión esto podría verse como:

```
In [201]: %%timeit
          result = [(x if c else y) for x, y, c in zip(xarr, yarr, cond)]
          len(result)

1.79 s ± 8.37 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Esto tiene múltiples problemas. Primero, no será muy rápido para arrays grandes (porque todo el trabajo se está realizando en el código de Python interpretado). En segundo lugar, no funcionará con matrices multidimensionales. Con `np.where` podemos escribir esto de forma concisa:

```
In [202]: %%timeit
          result = np.where(cond, xarr, yarr)
          len(result)

49.3 ms ± 160 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

El segundo y tercer argumento de `where` no necesitan ser matrices, uno o ambos pueden ser escalares. Incluso pueden omitirse, en cuyo caso lo que se obtiene son los índices de los elementos que cumplen la condición:

```
In [203]: arr = np.array([8, 8, 3, 7, 7, 0, 4, 2, 5, 2])
          index_cond = np.where(arr > 5)
          index_cond
```

```
Out[203]: (array([0, 1, 3, 4]),)
```

Una vez que tenemos las posiciones es posible extraer los elementos utilizando el método `take` :

```
In [204]: arr.take(index_cond) # equivalente arr[index_cond]
```

```
Out[204]: array([[8, 8, 7, 7]])
```

Un uso típico en análisis de datos es producir una nueva matriz de valores basados en otra matriz. Supongamos que tenemos una matriz de datos generados aleatoriamente y se desea reemplazar todos los valores positivos con 2 y todos los valores negativos con -2. Esto es muy fácil de hacer con `np.where` :

```
In [205]: arr = np.random.randn(4, 4)
arr
```

```
Out[205]: array([[ -0.35, -0.98,  1.93,  0.53],
 [ 1.58,  0.24,  0.4 ,  0.53],
 [-0.12,  0.31, -0.2 , -1.33],
 [-1.26,  0.71, -1.04,  1.48]])
```

```
In [206]: arr > 0
```

```
Out[206]: array([[False, False,  True,  True],
 [ True,  True,  True,  True],
 [False,  True, False, False],
 [False,  True, False,  True]])
```

```
In [207]: np.where(arr > 0, 2, -2)
```

```
Out[207]: array([[ -2, -2,  2,  2],
 [ 2,  2,  2,  2],
 [-2,  2, -2, -2],
 [-2,  2, -2,  2]])
```

Se pueden combinar valores escalares y matrices al usar `np.where`. Por ejemplo, se puede reemplazar todos los valores positivos en `arr` con la constante 2 así:

```
In [208]: np.where(arr > 0, 2, arr) # fija solo los valores positivos a 2
```

```
Out[208]: array([[ -0.35, -0.98,  2. ,  2. ],
 [ 2. ,  2. ,  2. ,  2. ],
 [-0.12,  2. , -0.2 , -1.33],
 [-1.26,  2. , -1.04,  2. ]])
```

## Métodos Estadísticos y Matemáticos

Numpy proporciona un conjunto de funciones matemáticas y estadísticas aplicables sobre una matriz completa o sobre los datos de un eje como métodos de la clase de `ndarray` y como funciones Numpy. Entre ellas están las agregaciones (a menudo llamadas reducciones) como `sum`, `mean` y `std` (desviación estándar).

En el siguiente ejemplo se generan datos aleatorios normalmente distribuidos y se computan sobre ellos algunas estadísticas agregadas:

```
In [209]: arr = np.random.randn(5, 4)
arr
```

```
Out[209]: array([[ 0.5 ,  0.68,  1.5 ,  0.18],
 [ 0.65,  0.13,  0.15, -0.25],
 [-0.58, -1.2 ,  0.54, -0.43],
 [-0.97,  1.84,  1.17, -0.04],
 [-0.15, -1.25, -0.55,  0.18]])
```

```
In [210]: arr.mean()
```

```
Out[210]: 0.10476301344690833
```

```
In [211]: np.mean(arr)
```

```
Out[211]: 0.10476301344690833
```

```
In [212]: arr.sum()
```

```
Out[212]: 2.0952602689381665
```

Cuando se llama a un método estadístico en una matriz booleana, `True` y `False` se convierten en 1 y 0 respectivamente. Esta propiedad se puede usar para contar la cantidad de elementos (casos) que satisfacen una determinada condición:

```
In [213]: x = np.random.randint(10, size=1000) # 1000 enteros aleatorios en el rango 0~9.
          (x > 5).sum() # cuenta el número de elementos mayores que 5
```

```
Out[213]: 429
```

Algunas funciones como `mean` y `sum` tiene un argumento de eje opcional que calcula la estadística sobre el eje dado, dando como resultado una matriz con una dimensión menor:

```
In [214]: arr.mean(axis=1)
```

```
Out[214]: array([ 0.71,  0.17, -0.42,  0.5 , -0.44])
```

```
In [215]: arr.sum(axis=0)
```

```
Out[215]: array([-0.54,  0.21,  2.8 , -0.37])
```

Otros métodos como `cumsum` y `cumprod` no generan valores agregados, en su lugar producen una matriz con los resultados intermedios:

```
In [216]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
          arr.cumsum()
```

```
Out[216]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

En matrices multidimensionales, las funciones de acumulación como `cumsum` devuelven una matriz del mismo tamaño, pero con los agregados parciales a lo largo de un eje los cálculos se realizan en base a los segmentos dimensionales inferiores:



```
In [217]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
          arr

Out[217]: array([[0, 1, 2],
                [3, 4, 5],
                [6, 7, 8]])

In [218]: arr.cumsum(axis=0)

Out[218]: array([[ 0,  1,  2],
                [ 3,  5,  7],
                [ 9, 12, 15]])

In [219]: arr.cumsum(axis=1)

Out[219]: array([[ 0,  1,  3],
                [ 3,  7, 12],
                [ 6, 13, 21]])
```

La siguiente tabla muestra los métodos estadísticos básicos aplicables a matrices:

Método	Descripción
sum	Suma todos los elementos de un array o a lo largo de un eje, los arrays vacíos suman 0
mean	Media aritmética, los arrays vacíos tiene media NaN
std, var	Desviación estándar y varianza, admiten el ajuste de los grados de libertad
min, max	Valores mínimos y máximos
argmin, argmax	Índices de los valores mínimos y máximos
cumsum	Suma acumulativa de los elementos empezando por 0
cumprod	Producto acumulativo de los elementos empezando por 1

Métodos para Arrays booleanos

Las variables booleanas están asignadas a los valores 1 ( True ) y 0 ( False ) en los métodos anteriores. Por lo tanto, la suma se usa a menudo como un medio para contar los valores verdaderos en una matriz booleana:

```
In [220]: arr = np.random.randn(100)
          (arr > 0).sum() # Número de valores positivos

Out[220]: 47
```

Hay dos métodos adicionales, all y any , útiles especialmente para matrices booleanas. any comprueba si uno o más valores en una matriz son True , mientras que all comprueb si todos los valores son True :

```
In [221]: bools = np.array([False, False, True, False])
          bools.any()

Out[221]: True

In [222]: bools.all()

Out[222]: False
```

**Nota:** Estos métodos también funcionan con arrays no booleanos, en los que los elementos distintos de 0 se evalúan a `True`.

## Ordenación

Al igual que las listas de Python, los arrays de Numpy pueden ser ordenados *internamente* ( `in-place` ) con el método `sort`. Es importante utilizar este método y no la función incluida en Python por cuestiones de eficiencia. Por defecto `np.sort` utiliza el algoritmo de clasificación rápida `quicksort`, aunque también están disponibles `mergesort` y `heapsort`. Para la mayoría de las aplicaciones, la ordenación rápida predeterminada es más que suficiente y tiene una complejidad  $O[N \log N]$ .

**Nota:** Si la matriz a ordenar es una vista de una matriz diferente, la matriz original se modificará.

```
In [223]: arr = np.random.randn(6)
          arr
```

```
Out[223]: array([-0.7 , -2.1 , -0.13, -0.6 , -0.07, -0.73])
```

```
In [224]: arr.sort()
          arr
```

```
Out[224]: array([-2.1 , -0.73, -0.7 , -0.6 , -0.13, -0.07])
```

Se puede ordenar cada sección unidimensional de valores en una matriz multidimensional pasando el número de eje para ordenar:

```
In [225]: arr = np.random.randn(5, 3)
          arr
```

```
Out[225]: array([[ -0.51, -0.37, -0.67],
                 [-1.72, -0.5 , -0.13],
                 [ 1.3 ,  0.78,  0.27],
                 [ 0.97, -1.11,  0.03],
                 [ 0.97,  1.22, -0.73]])
```

```
In [226]: arr.sort(1)
          arr
```

```
Out[226]: array([[ -0.67, -0.51, -0.37],
                 [-1.72, -0.5 , -0.13],
                 [ 0.27,  0.78,  1.3 ],
                 [-1.11,  0.03,  0.97],
                 [-0.73,  0.97,  1.22]])
```

La función Numpy `sort` devuelve una copia ordenada de una matriz en lugar de modificarla in-situ. Esta función es útil ya que no siempre nos interesa que una matriz quede ordenada.

```
In [227]: arr = np.random.randn(5)
          np.sort(arr)
```

```
Out[227]: array([-0.75, -0.05,  0.18,  0.79,  1.02])
```

```
In [228]: arr
Out[228]: array([-0.75, -0.05,  1.02,  0.79,  0.18])
```

Una forma rápida y de calcular los cuantiles de una matriz es ordenarla y seleccionar el valor en un rango particular:

```
In [229]: large_arr = np.random.randn(1000)
          np.sort(large_arr)[int(0.05 * len(large_arr))] # quantil de 5%
Out[229]: -1.6934425819669507
```

La función `argsort` vinculada a la ordenación de matrices, devuelve los índices de los elementos ordenados:

```
In [30]: x = np.array([2, 1, 4, 3, 5])
          i = np.argsort(x)
          print(i)
[1 0 3 2 4]
```

La ordenación puede hacerse por filas o columnas haciendo uso del parámetro `axis` :

```
In [31]: rand = np.random.RandomState(42)
          X = rand.randint(0, 10, (4, 6))
          print(X)
[[6 3 7 4 6 9]
 [2 6 7 4 3 7]
 [7 2 5 4 1 7]
 [5 1 4 0 9 5]]
```

```
In [32]: # ordenación por columnas
          np.sort(X, axis=0)
```

```
Out[32]: array([[2, 1, 4, 0, 1, 5],
                [5, 2, 5, 4, 3, 7],
                [6, 3, 7, 4, 6, 7],
                [7, 6, 7, 4, 9, 9]])
```

**Nota:** Utilizar este parámetro trata cada fila o columna como una matriz independiente, perdiendo cualquier relación entre los valores de fila o columna previamente existente.

A veces no estamos interesados en ordenar la matriz completa, simplemente queremos encontrar los K valores más pequeños en la matriz. NumPy proporciona la función `np.partition` que toma una matriz y un número K dando como resultado una nueva matriz con K los valores más pequeños a la izquierda de la partición y los valores restantes a la derecha, todos en orden arbitrario:

```
In [33]: x = np.array([7, 2, 3, 1, 6, 5, 4])
          np.partition(x, 3)
Out[33]: array([2, 1, 3, 4, 6, 5, 7])
```

También existe la correspondiente función `np.argpartition` que devuelve los índices de los valores.

### Unique y otras lógicas de conjuntos

NumPy tiene algunas operaciones básicas de conjuntos para ndarrays unidimensionales. Uno de uso común es `np.unique` , que devuelve los valores únicos ordenados en una matriz:

```
In [230]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
          np.unique(names)

Out[230]: array(['Bob', 'Joe', 'Will'], dtype='<U4')
```

```
In [231]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
          np.unique(ints)

Out[231]: array([1, 2, 3, 4])
```

El método `unique` permite recuperar la cuenta de cada item si es necesario:

```
In [232]: np.unique(ints, return_counts=True)

Out[232]: (array([1, 2, 3, 4]), array([2, 2, 3, 2]))
```

Otra función, `np.in1d` , comprueba la pertenencia de los valores en una matriz en otra, devolviendo una matriz booleana:

```
In [233]: values = np.array([6, 0, 0, 3, 2, 5, 6])
          np.any(np.in1d(values, [2, 3, 6]))

Out[233]: True
```

La siguiente tabla muestra las funciones aplicables a conjuntos:

Método	Descripción
<code>unique(x)</code>	Devuelve un array ordenado con los elementos únicos de <code>x</code>
<code>intersect1d(x,y)</code>	Devuelve un array ordenado con los elementos presentes en <code>x</code> e <code>y</code>
<code>union1d(x, y)</code>	Devuelve un array ordenado con los elementos de <code>x</code> e <code>y</code>
<code>in1d(x, y)</code>	Devuelve un array booleano indicando si cada elemento de <code>x</code> está contenido en <code>y</code>
<code>setdiff1d(x, y)</code>	Devuelve un array ordenado con los elementos presentes en <code>x</code> y no en <code>y</code>
<code>setxor1d(x, y)</code>	Devuelve un array ordenado con los elementos presentes en <code>x</code> y en <code>y</code> , pero no en ambos

### Salvado y recuperación de Arrays en disco

NumPy puede guardar y cargar datos desde y hacia el disco en formato de texto o binario. En esta sección, solo se analiza el formato binario incorporado de NumPy, ya que la mayoría de los usuarios preferirán `pandas` y otras herramientas para cargar texto o datos tabulares.

Las funciones `np.save` y `np.load` permiten guardar y cargar de forma eficiente los datos de una matriz en el disco. Las matrices se guardan de forma predeterminada en un formato binario en crudo ( `raw` ) sin comprimir con la extensión de archivo `.npy` (si no se incluye la extensión esta se añade por defecto):

```
In [234]: arr = np.arange(10)
          np.save('./data/out/some_array', arr)
```

```
In [235]: np.load('./data/out/some_array.npy')
```

```
Out[235]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Se pueden guardar múltiples arrays en un archivo sin comprimir usando `np.savez` y pasando los arrays como argumentos de palabras clave:

```
In [236]: np.savez('./data/out/array_archive.npz', a=arr, b=arr)
```

Al cargar un archivo `.npz`, se recupera un objeto similar a un diccionario que permite la carga de las matrices individualmente:

```
In [237]: arch = np.load('./data/out/array_archive.npz')
          arch['b']
```

```
Out[237]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Si los datos de una matriz son susceptibles de comprimirse correctamente, es posible utilizar la función `numpy.savez_compressed` en su lugar:

```
In [238]: np.savez_compressed('./data/out/arrays_compressed.npz', a=arr, b=arr)
```

```
In [239]: # !del en sistemas operativos Windows
          !rm some_array.npy
          !rm array_archive.npz
          !rm arrays_compressed.npz
```

```
rm: some_array.npy: No such file or directory
rm: array_archive.npz: No such file or directory
rm: arrays_compressed.npz: No such file or directory
```

Una forma estándar de importar conjuntos de datos es usar la función `np.genfromtxt` que puede trabajar con URL web, manejar valores faltantes, múltiples delimitadores, manejar un número irregular de columnas, etc. Una versión menos versátil es el `np.loadtxt` que asume que el conjunto de datos no tiene valores faltantes (ver [datos faltantes](#)).

```
In [240]: # sin notación científica
          np.set_printoptions(suppress=True)

          # Importa los datos desde un fichero CSV
          path = './data/teoria/autos.csv'
          data = np.genfromtxt(path, delimiter=',', skip_header=1, filling_values=-999, dtype='float')
          data[:3] # se ven las 3 primeras filas
```

```
Out[240]: array([[ 18. ,   8. ,  307. ,  130. , 3504. ,   12. ,   70. ,   1. ,
                  -999. ],
                 [ 15. ,   8. ,  350. ,  165. , 3693. ,  11.5,   70. ,   1. ,
                  -999. ],
                 [ 18. ,   8. ,  318. ,  150. , 3436. ,   11. ,   70. ,   1. ,
                  -999. ]])
```

Dado que todos los elementos en una matriz numpy deben ser del mismo tipo de datos, la última columna que es un texto se importará como un `np.nan` de forma predeterminada. Al establecer el argumento `filling_values` puede reemplazar los valores faltantes con otra cosa.

En el caso, de tener que recuperar la columna de texto tal como está sin reemplazarla con un marcador de posición, se puede establecer el tipo de `dtype` como `object` o como `None`.

```
In [241]: data = np.genfromtxt(path, delimiter=',', skip_header=1, dtype='object', encoding='ascii')
          data[:3] # se ven las 3 primeras filas
```

```
Out[241]: array([[b'18', b'8', b'307', b'130', b'3504', b'12', b'70', b'1',
                  b"chevrolet chevelle malibu"],
                 [b'15', b'8', b'350', b'165', b'3693', b'11.5', b'70', b'1',
                  b"buick skylark 320"],
                 [b'18', b'8', b'318', b'150', b'3436', b'11', b'70', b'1',
                  b"plymouth satellite"]], dtype=object)
```

```
In [242]: data = np.genfromtxt(path, delimiter=',', skip_header=1, dtype=None, encoding='ascii')
          data[:3] # se ven las 3 primeras filas
```

```
Out[242]: array([(18., 8, 307., 130, 3504, 12., 70, 1, "chevrolet chevelle malibu"),
                 (15., 8, 350., 165, 3693, 11.5, 70, 1, "buick skylark 320"),
                 (18., 8, 318., 150, 3436, 11., 70, 1, "plymouth satellite")],
                 dtype=[('f0', '<f8'), ('f1', '<i8'), ('f2', '<f8'), ('f3', '<i8'), ('f4', '<i8'), ('f5', '<f8'), ('f6', '<i8'), ('f7', '<i8'), ('f8', '<U38')])
```

Por último, para exportar la información de un array a un fichero CSV está disponible el método `savetxt`:

```
In [243]: # Salva el array como un fichero CSV, al ser de dtype None es necesario fijar el formato como texto
          np.savetxt('./data/out/autos_out.csv', data, delimiter=",", fmt='%s')
```

## Ficheros mapeados en memoria

Un archivo mapeado en memoria es un método para interactuar con datos binarios en el disco como si estuviera almacenado en una matriz en memoria. NumPy implementa el objeto `memmap` que es similar a `ndarray`, permitiendo que pequeños segmentos de un archivo grande se lean y escriban sin leer toda la matriz en la memoria.

Además, un `memmap` tiene los mismos métodos que una matriz en memoria y, por lo tanto, se puede sustituir en muchos algoritmos donde se esperaría un `ndarray`. Tienen, por tanto, la ventaja adicional de permitir trabajar con conjuntos de datos que no se pueden alojar en la memoria RAM por su tamaño.

```
In [244]: # fichero mapeado en memoria de unos 800 Mb
          mmap = np.memmap('./data/out/mi_mapa.dat', dtype='float64', mode='w+', shape=(10000, 10000))
          mmap
```

```
Out[244]: memmap([[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.]])
```

Segmentar un `memmap` devuelve una vista de los datos en el disco:

```
In [245]: section = mmap[:5]
          section
```

```
Out[245]: memmap([[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.]])
```

Si asignamos datos a segmentos de un `memmap` éstos se almacenarán en la memoria (como un objeto de archivo Python), pero podremos escribirlos en el disco llamando al método `flush` :

```
In [246]: section[:] = np.random.randint(1, 5, size=10000)
          mmap
```

```
Out[246]: memmap([[4., 2., 3., ..., 2., 3., 1.],
                  [4., 2., 3., ..., 2., 3., 1.],
                  [4., 2., 3., ..., 2., 3., 1.],
                  ...,
                  [0., 0., 0., ..., 0., 0., 0.],
                  [0., 0., 0., ..., 0., 0., 0.],
                  [0., 0., 0., ..., 0., 0., 0.]])
```

```
In [247]: del mmap
          !rm ./data/out/mi_mapa.dat
```

Cada vez que un mapa de memoria queda fuera del alcance y se recolecta la basura, los cambios también se actualizarán en el disco. Al abrir un mapa de memoria existente, todavía debe especificar el tipo y la forma, ya que el archivo es solo un bloque de datos binarios sin metadatos en el disco.

## Generación de número pseudoaleatorios

El módulo `numpy.random` complementa el Python `random` con funciones para generar de manera eficiente matrices completas de valores de muchos tipos de distribuciones de probabilidad.

Por ejemplo, puede obtener una matriz  $4 \times 4$  de muestras de la distribución normal estándar usando `normal` :

```
In [248]: samples = np.random.normal(size=(4, 4))
          samples
```

```
Out[248]: array([[ 0.94,  0.02, -0.53, -1.44],
                  [ 0.64, -1.75, -0.12,  1.77],
                  [ 0.26, -0.34, -0.14, -1.34],
                  [-1.82, -0.65, -0.14, -1.26]])
```

El módulo aleatorio incorporado de Python, por el contrario, solo muestrea un valor a la vez. Como se puede ver en este ejemplo, `numpy.random` es al menos un orden de magnitud más rápido para generar muestras muy grandes:

```
In [249]: from random import normalvariate
          N = 1000000
          %timeit samples = [normalvariate(0, 1) for _ in range(N)]
          %timeit np.random.normal(size=N)
```

```
834 ms ± 7.37 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
35.3 ms ± 307 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Estos son números pseudoaleatorios porque son generados por un algoritmo con comportamiento determinista basado en la semilla del generador de números aleatorios de NumPy. Se puede cambiar la semilla de generación de números aleatorios de NumPy usando `np.random.seed` :

```
In [250]: np.random.seed(1234)
```

Las funciones de generación de datos en `numpy.random` utilizan una semilla aleatoria global. Para evitar el estado global , puede usar `numpy.random.RandomState` para crear un generador de números aleatorios aislado de otros:

```
In [251]: rng = np.random.RandomState(1234)
          rng.randn(10)
```

Out[251]: array([ 0.47, -1.19, 1.43, -0.31, -0.72, 0.89, 0.86, -0.64, 0.02,
 -2.24])

La siguiente tabla algunas de las funciones mas comunes de `numpy.random` :

Función	Descripción
seed	Modifica la semilla del generador de números aleatorios
permutation	Devuelve una permutación aleatoria de una secuencia, o devuelve un rango permutado
shuffle	Permuta aleatoriamente una secuencia in-situ
rand	Genera muestras de una distribución uniforme
randint	Genera números enteros aleatorios dentro de de un rango dado
randn	Genera muestras de una distribución normal con media 0 y desviación estándar 1
binomial	Genera muestras de una distribución binomial
normal	Genera muestras de una distribución normal (Gausiana)
beta	Genera muestras de una distribución beta
chisquare	Genera muestras de una distribución chi-cuadrado
gamma	Genera muestras de una distribución gamma
uniform	Genera muestras de una distribución uniforme [0,1)

Es muy útil el método `choice` que genera valores aleatorios utilizando los valores incluidos en un array unidimensional, permitiendo modificar la probabilidad de los elementos:

```
In [252]: np.random.choice(['a', 'e', 'i', 'o', 'u'], size=10)
```

Out[252]: array(['o', 'u', 'u', 'a', 'e', 'e', 'e', 'i', 'o', 'u'], dtype='<U1')

```
In [253]: np.random.choice(['a', 'e', 'i', 'o', 'u'], size=10, p=[0.3, 0.1, 0.1, 0.4, 0.1])
```

Out[253]: array(['a', 'a', 'a', 'i', 'e', 'o', 'a', 'a', 'i', 'a'], dtype='<U1')



## Arrays Estructurados (registros)

Un `ndarray` es un contenedor de **datos homogéneo**, es decir, representa un bloque de memoria en el que cada elemento ocupa el mismo número de bytes, determinado por el tipo de `dtype`. A priori, esto parece no permitir representar datos heterogéneos o de tipo tabular. Una matriz estructurada es un `ndarray` en el que se puede pensar que cada elemento representa una estructura en C (de ahí el nombre "estructurado") o una fila de una tabla SQL con varios campos.

Hay varias formas de especificar un tipo de estructurado. Una forma típica es definir una lista de tuplas con formato `(nombre_de_campo, tipo_de_datos_de_campo)`:

**Nota:** Para mayor claridad, los tipos numéricos se pueden especificar con tipos de Python o dtypes NumPy. El primer carácter (opcional) es `<` o `>`, que significa "little endian" o "big endian", respectivamente, y especifica la convención de ordenación de los bits significativos.

Caracter	Descripción	Ejemplo
'b'	Byte	<code>np.dtype('b')</code>
'i'	Signed integer	<code>np.dtype('i4') == np.int32</code>
'u'	Unsigned integer	<code>np.dtype('u1') == np.uint8</code>
'f'	Floating point	<code>np.dtype('f8') == np.int64</code>
'c'	Complex floating point	<code>np.dtype('c16') == np.complex128</code>
'S', 'a'	string	<code>np.dtype('S5')</code>
'U'	Unicode string	<code>np.dtype('U') == np.str_</code>
'V'	Raw data (void)	<code>np.dtype('V') == np.void</code>

```
In [254]: xy_dtype = [('x', np.float64), ('y', np.int32)]
          sarr = np.array([(1.5, 6), (np.pi, -2)], dtype=xy_dtype)
          sarr
```

Out[254]: array([(1.5 , 6), (3.14, -2)], dtype=[('x', '<f8'), ('y', '<i4')])

Se pueden crear arrays estructurados pasando al parámetro `dtype` un diccionario con las claves `names` y `formats`:

```
In [4]: sarr = np.zeros(4, dtype={'names': ('x', 'y', 'z'), 'formats': ('U10', 'i4', 'f8')})
          sarr
```

Out[4]: array([(' ', 0, 0.), (' ', 0, 0.), (' ', 0, 0.), (' ', 0, 0.)],
 dtype=[('x', '<U10'), ('y', '<i4'), ('z', '<f8')])

Para reutilizar tipos estructurados se puede hacer uso de la función `dtype`:

```
In [5]: stype = np.dtype({'names': ('x', 'y', 'z'), 'formats': ('U10', 'i4', 'f8')})
          sarr = np.zeros(4, dtype=stype)
          sarr
```

Out[5]: array([(' ', 0, 0.), (' ', 0, 0.), (' ', 0, 0.), (' ', 0, 0.)],
 dtype=[('x', '<U10'), ('y', '<i4'), ('z', '<f8')])

Si los nombres de los tipos no le importan, puede especificar los tipos solos en una cadena separada por comas:

```
In [10]: stype = np.dtype('U10, i4, f8')
         stype
```

```
Out[10]: dtype([('f0', '<U10'), ('f1', '<i4'), ('f2', '<f8')])
```

Los elementos de la matriz son objetos similares a tuplas a los que se puede acceder como un diccionario:

```
In [17]: sarr[0] = ('eje1', 3, 9.7)
         sarr[1] = ('eje2', 5, 1.0)
         sarr[2] = ('eje3', 7, 3)
```

```
In [18]: sarr[0]['y']
```

```
Out[18]: 3
```

Los nombres de los campos se almacenan en el atributo `dtype.names`. Cuando accede a un campo en la matriz estructurada, se devuelve una vista de los datos, no se copia nada:

```
In [19]: sarr['x']
```

```
Out[19]: array(['eje1', 'eje2', 'eje3', ''], dtype='<U10')
```

Al especificar un `dtype` estructurado, también puede pasar un `shape` (como entero o tupla):

```
In [258]: xy_dtype = [('x', np.int64, 3), ('y', np.int32)]
         sarr = np.zeros(4, dtype=xy_dtype)
         sarr
```

```
Out[258]: array([([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0)],
              dtype=[('x', '<i8', (3,)), ('y', '<i4')])
```

Se pueden expresar estructuras más complicadas y anidadas como un solo bloque de memoria en una matriz. También se pueden anidar `dtypes` para hacer estructuras más complejas:

```
In [259]: xy_dtype = [('x', [('a', 'f8'), ('b', 'f4')]), ('y', np.int32)]
         sarr = np.array([(1, 2), 5], [(3, 4), 6], dtype=xy_dtype)
         sarr['x']
```

```
Out[259]: array([(1., 2.), (3., 4.)], dtype=[('a', '<f8'), ('b', '<f4')])
```

```
In [260]: sarr['x']['a']
```

```
Out[260]: array([1., 3.])
```

Comparados con los `DataFrame` de `pandas`, los arrays estructurados de `NumPy` son una herramienta comparativamente de bajo nivel. Proporcionan un medio para interpretar un bloque de memoria como una estructura tabular con columnas anidadas arbitrariamente complejas. Dado que cada elemento de la matriz se representa en la memoria como un número fijo de bytes, las matrices estructuradas proporcionan una forma muy rápida y eficiente de escribir datos en y desde el disco (incluidos los mapas de memoria), transportarlos a través de la red y otros usos similares.

Un uso común para arrays estructurados es escribir archivos de datos como flujos de bytes con registros de longitud fija. Esta es una forma común de serializar datos en código C y C++, que se encuentra comúnmente en sistemas heredados en la industria. Siempre que se conozca el formato del archivo (el tamaño de cada registro y el orden, tamaño de byte y tipo de datos de cada elemento), los datos se pueden leer en la memoria con `np.fromfile`.

## Datetimes y Timedeltas

A partir de NumPy 1.7, existen tipos de datos de matriz de núcleo que admiten de forma nativa la funcionalidad de datetime. El tipo de datos se llama "datetime64", llamado así porque "datetime" ya está en uso por la biblioteca datetime incluida en Python.

### Fechas básicas

La forma más básica de crear tiempos de datos es a partir de cadenas en formato de fecha o fecha de la norma ISO 8601. La unidad para almacenamiento interno se selecciona automáticamente de la forma de la cadena y puede ser una unidad de fecha o una unidad de tiempo. Las unidades de fecha son años ('Y'), meses ('M'), semanas ('W') y días ('D'), mientras que las unidades de tiempo son horas ('h'), minutos ('m'), segundos ('s'), milisegundos ('ms') y algunas unidades basadas en segundos con prefijos adicionales: us-microsegundos, ns-nanosegundos, ps-picosegundos...

```
In [261]: # Fecha ISO simple
          np.datetime64('2019-02-25')
```

```
Out[261]: numpy.datetime64('2019-02-25')
```

```
In [262]: # Utilizando meses
          np.datetime64('2019-02')
```

```
Out[262]: numpy.datetime64('2019-02')
```

```
In [263]: # Utilizando fechas y horas
          np.datetime64('2012-02-25T03:30')
```

```
Out[263]: numpy.datetime64('2012-02-25T03:30')
```

Al crear una matriz de tiempos de datos a partir de una cadena, aún es posible seleccionar automáticamente la unidad de las entradas, utilizando el tipo datetime con unidades genéricas.

```
In [264]: np.array(['2019-07-13', '2019-01-13', '2019-08-13'], dtype='datetime64')
```

```
Out[264]: array(['2019-07-13', '2019-01-13', '2019-08-13'], dtype='datetime64[D]')
```

El tipo datetime funciona con muchas funciones NumPy comunes, por ejemplo, `arange` se puede usar para generar rangos de fechas.

```
In [265]: np.arange('2019-02', '2019-03', dtype='datetime64[D]')
```

```
Out[265]: array(['2019-02-01', '2019-02-02', '2019-02-03', '2019-02-04',
                  '2019-02-05', '2019-02-06', '2019-02-07', '2019-02-08',
                  '2019-02-09', '2019-02-10', '2019-02-11', '2019-02-12',
                  '2019-02-13', '2019-02-14', '2019-02-15', '2019-02-16',
                  '2019-02-17', '2019-02-18', '2019-02-19', '2019-02-20',
                  '2019-02-21', '2019-02-22', '2019-02-23', '2019-02-24',
                  '2019-02-25', '2019-02-26', '2019-02-27', '2019-02-28'],
                  dtype='datetime64[D]')
```

El objeto datetime representa un solo momento en el tiempo. Si dos tiempos de datos tienen unidades diferentes, aún pueden representar el mismo momento del tiempo, y la conversión de una unidad más grande como meses a una unidad más pequeña como los días se considera un lanzamiento "seguro" porque el momento del tiempo todavía se está representando exactamente.

```
In [266]: np.datetime64('2019') == np.datetime64('2019-01-01')
```

```
Out[266]: True
```

## Fecha y Aritmética Timedelta

NumPy permite la resta de dos valores de fecha y hora, una operación que produce un número con una unidad de tiempo. Debido a que NumPy no tiene un sistema de cantidades físicas en su núcleo, el tipo de datos `timedelta64` se creó para complementar `datetime64`.

Datetimes y Timedeltas trabajan juntos para proporcionar formas para cálculos simples de fecha y hora.

```
In [267]: np.datetime64('2009-01-01') - np.datetime64('2008-01-01')
```

```
Out[267]: numpy.timedelta64(366, 'D')
```

```
In [268]: np.datetime64('2009') + np.timedelta64(20, 'D')
```

```
Out[268]: numpy.datetime64('2009-01-21')
```

```
In [269]: np.datetime64('2011-06-15T00:00') + np.timedelta64(12, 'h')
```

```
Out[269]: numpy.datetime64('2011-06-15T12:00')
```

```
In [270]: np.timedelta64(1, 'W') / np.timedelta64(1, 'D')
```

```
Out[270]: 7.0
```

```
In [271]: np.timedelta64(1, 'W') % np.timedelta64(10, 'D')
```

```
Out[271]: numpy.timedelta64(7, 'D')
```

Hay dos unidades de `Timedelta` ("Y", años y "M", meses) que se tratan especialmente, porque la cantidad de tiempo que representan los cambios depende de cuándo se utilizan. Si bien una unidad de días de `Timedelta` equivale a 24 horas, no hay forma de convertir una unidad de mes en días, porque los diferentes meses tienen diferentes números de días.

```
In [272]: a = np.timedelta64(1, 'Y')
          np.timedelta64(a, 'M')
```

```
Out[272]: numpy.timedelta64(12, 'M')
```

```
In [273]: np.timedelta64(a, 'D')
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-273-26823ba00a80> in <module>
----> 1 np.timedelta64(a, 'D')

TypeError: Cannot cast NumPy timedelta64 scalar from metadata [Y] to [D] according to the rule 'same_kind'
```

## Funcionalidad de día hábil

Para permitir que la fecha y la hora se utilicen en contextos donde solo ciertos días de la semana son válidos, NumPy incluye un conjunto de funciones de "día de trabajo" (*business day*).

Para gestionar los días hábiles se mantiene un calendario de días hábiles (*busdaycalendar*) que almacena de manera eficiente información que define los días válidos para la familia de funciones de los días hábiles. Los días válidos predeterminados son de lunes a viernes ("días hábiles"). Se puede definir un objeto *busdaycalendar* con cualquier conjunto de días válidos semanales, además de fechas de "vacaciones" opcionales que siempre serán inválidas.

La implementación se basa en una "máscara de semana" que contiene 7 banderas booleanas para indicar días válidos (de lunes a domingo). Una vez que se crea un objeto *busdaycalendar*, la máscara de la semana y los días festivos no se pueden modificar.

```
In [288]: # Fiestas de Santo Tomás y de la ETSISI
fiestas = np.array(['2019-01-28', '2019-01-29'], dtype='datetime64[D]')
# Los 'lunes al sol'
bdd = np.busdaycalendar(weekmask='0111100', holidays=fiestas)
# El valor predeterminado es de lunes a viernes
print(bdd.weekmask)
# Se eliminan los días festivos que ya sean de fin de semana
print(bdd.holidays)

[False  True  True  True  True False False]
['2019-01-29']
```

La función `busday_offset` le permite aplicar las compensaciones especificadas en días hábiles a fechas de tiempo con una unidad de 'D' (día).

```
In [292]: np.busday_offset('2019-01-25', 1, busdaycal=bdd) # viernes
Out[292]: numpy.datetime64('2019-01-30')
```

Cuando una fecha de entrada cae en el fin de semana o en un día festivo, `busday_offset` primero aplica una regla para pasar la fecha a un día hábil válido en base al parámetro `roll` y luego aplica la compensación. La regla predeterminada de `roll` es "raise", que simplemente genera una excepción. Las reglas más utilizadas son "hacia adelante" y "hacia atrás".

```
In [294]: np.busday_offset('2019-01-26', 2, busdaycal=bdd)

-----
ValueError                                Traceback (most recent call last)
<ipython-input-294-599b36b0b32c> in <module>
----> 1 np.busday_offset('2019-01-26', 1, busdaycal=bdd)

ValueError: Non-business day date in busday_offset
```

```
In [295]: np.busday_offset('2019-01-26', 2, roll='forward', busdaycal=bdd)
Out[295]: numpy.datetime64('2019-02-01')
```

```
In [296]: np.busday_offset('2019-01-26', 2, roll='backward', busdaycal=bdd)
Out[296]: numpy.datetime64('2019-01-31')
```

Para probar un valor de `datetime64` para ver si es un día válido, use `is_busday`.

```
In [278]: np.is_busday(np.datetime64('2019-07-19')) # viernes
```

```
Out[278]: True
```

```
In [279]: np.is_busday(np.datetime64('2019-07-20')) # sábado
```

```
Out[279]: False
```

Para encontrar cuántos días válidos hay en un rango específico de fechas `datetime64`, use `busday_count` :

```
In [280]: np.busday_count(np.datetime64('2019-07-19'), np.datetime64('2019-07-29'))
```

```
Out[280]: 6
```

```
In [281]: np.busday_count(np.datetime64('2019-07-29'), np.datetime64('2019-07-19'))
```

```
Out[281]: -6
```

Si tiene una matriz de valores de `datetime64` días y desea un conteo de cuántos de ellos son fechas válidas, puede hacer esto:

```
In [282]: a = np.arange(np.datetime64('2019-07-19'), np.datetime64('2019-07-29'))
          np.count_nonzero(np.is_busday(a))
```

```
Out[282]: 6
```

## Datos faltantes

El tratamiento de los datos faltantes o datos no proporcionados es esencial. En particular, a muchos conjuntos de datos interesantes les faltará cierta cantidad de datos. Para complicar aún más las cosas, diferentes fuentes de datos pueden indicar datos faltantes de diferentes maneras. Nos referiremos a los datos faltantes en general como `valores nulos` , `NaN` o `NA` .

La forma de indicar los valores faltantes sigue normalmente dos estrategias: usar una máscara que indique globalmente los valores perdidos o elegir un valor centinela que indique una entrada faltante. Ninguno de estos enfoques está exento de problemas: el uso de una matriz de máscara separada requiere la asignación de una matriz booleana adicional, que agrega gastos generales tanto en el almacenamiento como en el cálculo. Un valor centinela reduce el rango de valores válidos que se pueden representar y puede requerir lógica adicional (a menudo no optimizada) en la aritmética de CPU y GPU. Los valores especiales comunes como `NaN` no están disponibles para todos los tipos de datos.

La forma en que otros paquetes como Pandas maneja los valores perdidos está limitada por su dependencia del paquete NumPy, que no tiene una noción incorporada de valores `NA` para tipos de datos que no son de punto flotante. Por otra parte, aunque NumPy tiene soporte para matrices enmascaradas, es decir, matrices con máscara booleana separada adjunta para marcar datos como "buenos" o "malos", la sobrecarga tanto en almacenamiento, computación y mantenimiento de código hace que sea una elección poco atractiva.

## None

El primer valor centinela utilizado por Numpy es `None` , un objeto singleton de Python que se usa a menudo para los datos faltantes en el código de Python. Debido a que `None` es un objeto de Python, sólo se puede usar en matrices NumPy con el tipo de datos `object` . Su uso implica una sobrecarga en las operaciones a diferencia con el uso de tipos nativos:

```
In [3]: for dtype in ['object', 'int']:
        print("dtype =", dtype)
        %timeit np.arange(1E6, dtype=dtype).sum()
        print()
```

```
dtype = object
50.1 ms ± 336 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

dtype = int
1.08 ms ± 5.23 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

El uso de objetos Python en una matriz también significa que si realiza agregaciones como `sum` o `min` en una matriz con un valor `None`, generalmente obtendrá un error:

```
In [5]: a = np.array([1, 2, 3, None])
        print(a.dtype)
        a.sum()
```

```
object
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-5-2a77b9b16ca1> in <module>
      1 a = np.array([1, 2, 3, None])
      2 print(a.dtype)
----> 3 a.sum()

/opt/conda/lib/python3.8/site-packages/numpy/core/_methods.py in _sum(a, axis,
dtype, out, keepdims, initial, where)
     36 def _sum(a, axis=None, dtype=None, out=None, keepdims=False,
     37           initial=_NoValue, where=True):
----> 38     return umr_sum(a, axis, dtype, out, keepdims, initial, where)
     39
     40 def _prod(a, axis=None, dtype=None, out=None, keepdims=False,
```

**TypeError:** unsupported operand type(s) for +: 'int' and 'NoneType'

## NaN

La otra representación de datos faltantes, `NaN` (acrónimo de *Not a Number*), es diferente; es un valor de punto flotante especial reconocido por todos los sistemas que utilizan la representación de punto flotante estándar IEEE:

```
In [6]: a = np.array([1, np.nan, 3, 4])
        print(a.dtype)
        a.sum()
```

```
float64
```

```
Out[6]: nan
```

A diferencia del ejemplo anterior el tipo resultante es de tipo `float`, lo que significa que soportará operaciones más rápidas y operaciones aritméticas, pero teniendo en cuenta que el resultado de una operación aritmética con un `NaN` es otro `NaN`. NumPy funciones de agregación especiales para ignorar los valores faltantes como `np.nansum`, `np.nanmin` o `np.nanmax`.

```
In [8]: np.nansum(a)
```

```
Out[8]: 8.0
```

**Nota:** NaN es específicamente un valor de punto flotante, es decir, no hay equivalente del valor NaN para enteros, cadenas u otros tipos. Por otra parte, hay que tener en cuenta que en Python (y NumPy), los NaN no se comparan igual ( `np.nan != np.nan` ), pero None lo hace.

```
In [3]: print (None == None)
        print (np.nan == np.nan)
```

```
True
False
```

---