

Librería Numpy

Antonio Espín Herranz

Instalación

- Para instalar numpy:
pip install numpy

O de descargar el archivo **whl** de **pypi.org**

- Para probar que se ha instalado bien:
 - En una consola de Python

```
import numpy as np
dir(np) → listar el contenido del módulo
```

 - El alias **np** es el más habitual.
 - Si al importar no hay errores → está bien instalada
 - **No se aconseja importar así: `from numpy import *`**
 - Ya que hay muchas funciones que entrarían en conflicto con funciones definidas en Python: `min`, `max`, etc
- También forma parte del paquete de librerías instaladas por **conda**.

Numpy

- **Numerical Python:**

- Está librería está enfocada hacia la informática científica.
- <http://www.numpy.org/>

- **Proporciona:**

- Un objeto array multidimensional: **array** (rápido y eficiente).
- Funciones para realizar cálculos de elementos con matrices o matemática sin necesidad de escribir bucles adicionales.
- Operaciones de álgebra lineal, transformada de Fourier y generación de números aleatorios.
- Herramientas para la integración de código C, C++ y Fortran.

- **Convenciones:**

- Para importar la librería:
 - **import numpy as np**
 - **from numpy import ***

Enfoque de Numpy

- Numpy es importante para los cálculos numéricos en Python está diseñado de forma eficiente para grandes conjuntos de datos.
 - Numpy almacena los datos en bloques contiguos de memoria.
 - La biblioteca de algoritmos de Numpy están escritos en lenguaje C.
 - Las operaciones de Numpy realizan cálculos complejos en arreglos completos sin la necesidad de bucles for de Python.
 - Los algoritmos basados en Numpy son de 10 a 100 veces más rápidos o más

Comparar tiempos

- `import numpy as np`
- `my_arr = np.arange(1000000)`
- `my_list = list(range(1000000))`

- `for _ in range(10): my_arr2 = my_arr * 2`
- `for _ in range(10): my_list2 = [x*2 for x in my_list]`

Como se trabaja con Numpy

```
In [12]: import numpy as np
```

```
# Generar datos aleatorios:
```

```
In [13]: data = np.random.randn(2, 3)
```

```
In [14]: data
```

```
Out[14]: array([[ -0.2047,  0.4789, -0.5194],  
                [-0.5557,  1.9658,  1.3934]])
```

```
# Luego realizamos operaciones como si fueran tipos escalares, no necesitamos bucles for para  
operar con los arrays.
```

```
In [15]: data * 10
```

```
Out[15]: array([[ -2.0471,  4.7894, -5.1944],  
                [-5.5573, 19.6578, 13.9341]])
```

```
In [16]: data + data
```

```
Out[16]: array([[ -0.4094,  0.9579, -1.0389],  
                [-1.1115,  3.9316,  2.7868]])
```

NumPy: Función array vs objeto ndarray

- Es el principal objeto de numpy **ndarray** que permite realizar operaciones con todos los elementos del array sin utilizar bucles (como ocurriría con una lista).
- Puede ser de una dimensión o de dos dimensiones, o de N-dimensiones.
- Se puede crear a partir de listas.
 - Utilizando la función **np.array** y otras más de la librería
- Ajusta el tipo según los elementos:
 - Por ejemplo: [9, 5.5, 10, 30] elige float.

```
>>> import numpy as np
>>> a = np.array(list(range(10)))
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> type(a)
<class 'numpy.ndarray'>
>>> b = np.arange(1,100,2)
>>> b
array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99])
>>> type(b)
<class 'numpy.ndarray'>
```

np.array es una **función** de numpy para crear **objetos ndarray**
Array n-dimensionales

Ejemplos

- `>>> np.array([1, 2, 3])`
- `array([1, 2, 3])`
- `>>> np.array([1, 2, 3.0])`
- `array([1., 2., 3.])`
- `>>> np.array([[1, 2], [3, 4]])`
- `array([[1, 2],[3, 4]])`
- `>>> np.array([1, 2, 3], ndmin=2)`
- `array([[1, 2, 3]])`
- `>>> np.array([1, 2, 3], dtype=complex)`
- `array([1.+0.j, 2.+0.j, 3.+0.j])`
- Podemos imprimir el tipo con la propiedad **dtype**
- `arr = np.array([8., 9., 0])`
- `print(arr.dtype)`
- La forma del array en filas y cols:
- `print(arr.shape)`
- Ver la dimension del array:
- `print(arr.ndim)`

NumPy: Tipos

Type	Type Code	Description
int8, uint8	i1, u1	Signed and unsigned 8-bit (1 byte) integer types
int16, uint16	i2, u2	Signed and unsigned 16-bit integer types
int32, uint32	i4, u4	Signed and unsigned 32-bit integer types
int64, uint64	i8, u8	Signed and unsigned 32-bit integer types
float16	f2	Half-precision floating point
float32	f4 or f	Standard single-precision floating point. Compatible with C float
float64, float128	f8 or d	Standard double-precision floating point. Compatible with C double and Python float object

NumPy: Tipos

Type	Type Code	Description
float128	f16 or g	Extended-precision floating point
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool	?	Boolean type storing True and False values
object	O	Python object type
string_	S	Fixed-length string type (1 byte per character). For example, to create a string dtype with length 10, use 'S10'.
unicode_	U	Fixed-length unicode type (number of bytes platform specific). Same specification semantics as string_ (e.g. 'U10').

Tipos en Numpy

- In [33]: `arr1 = np.array([1, 2, 3], dtype=np.float64)`
- In [34]: `arr2 = np.array([1, 2, 3], dtype=np.int32)`

- In [35]: `arr1.dtype`
- Out[35]: `dtype('float64')`

- In [36]: `arr2.dtype`
- Out[36]: `dtype('int32')`

Tipos en Numpy

- Los **dtypes** son una fuente de flexibilidad de NumPy para interactuar con datos provenientes de otros sistemas. En la mayoría de los casos, proporcionan un mapeo directamente en una representación de memoria o disco subyacente, lo que facilita la lectura y escritura de flujos binarios de datos en el disco y también la conexión al código escrito en un lenguaje de bajo nivel como C o Fortran.
- Los dtypes numéricos se denominan de la misma manera: un nombre de tipo, como float o int, seguido de un número que indica el número de bits por elemento

Ejemplos

- La conversión de tipos se puede hacer con el método **astype(tipo)**:
- In [41]: `arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])`
- In [42]: `arr`
- Out[42]: `array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])`
- In [43]: `arr.astype(np.int32)`
- Out[43]: `array([3, -1, -2, 0, 12, 10], dtype=int32)`

Ejemplos

- Si tenemos un array con string que representan números se pueden convertir a números:
 - In [44]: `numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)`
 - In [45]: `numeric_strings.astype(float)`
 - Out[45]: `array([1.25, -9.6 , 42.])`
- Si falla el casting, obtendremos una exception: `ValueError`

astype

- Siempre crea una nueva matriz (hace una copia).
- Podemos utilizar el tipo de otra matriz:
 - In [46]: `int_array = np.arange(10)`
 - In [47]: `calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)`
 - In [48]: `int_array.astype(calibers.dtype)`
 - Out[48]: `array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])`
- También se pueden utilizar abreviaturas para los tipos de numpy:
 - `empty_uint32 = np.empty(8, dtype='u4')`

Operaciones básicas: array

- **Métodos** para crear arrays:
 - `a = np.arange(1,7)`: Genera valores del 1 al 6
 - `print(np.arange(20))`: Del 0 al 19
 - `print(np.arange(5,10))`: Del 5 al 9
 - `print(np.zeros((2,3,4)))`: 2 tablas de 3 filas por 4 cols (de ceros)
 - `print(np.ones((2,3,4)))`: 2 tablas de 3 filas por 4 cols (de unos)
 - `print(np.empty((2,10)))`: 2 filas por 10 cols (vacías, saldrán números)
 - *OJO empty no tienen porque ser ceros, puede ser basura.*
 - `a.fill(10)` Rellena todos los elementos con 10
 - `a.astype(complex)`
 - Devuelve el array con los elementos convertidos a complex.
 - `a.astype(np.float64)`
 - Devuelve el array con los elementos convertidos a np.float64.
- **Propiedades:**
 - `print(a.size)`: Tamaño del array
 - `print(a.itemsize)`: Tamaño del item
 - `print(a.nbytes)`: Los bytes que ocupa

Resumen funciones de creación de matrices

- **array**

- Convertir entrada de datos (lista, tupla, matriz u otro tipo de secuencia a un **ndarray**, infiriendo el tipo o indicándolo en el parámetro **dtype**.

- **asarray**

- Convertir entrada a ndarray

- **arange**

- Como range de Python

- **ones, ones_like**

- Genera matrices de unos, y ones_like toma de ejemplo otra matriz.

- **zeros, zeros_like**

- Igual que la anterior pero con ceros

Resumen funciones de creación de matrices

- **empty, empty_like**

- Crea la matriz igual que los dos anteriores pero no tienen porque ser ni unos ni ceros.

- **eye, identity**

- Crea una matriz identidad, con unos en la diagonal principal y el resto a ceros.

- **full, full_like**

- `b = np.full((4,5), fill_value=9)`
 - Crea una matriz de 4 x 5 rellena de 9s.
- `a = np.full_like(b, fill_value=5)`
 - Toma de muestra otra matriz 'b', y la genera del mismo tipo y tamaño rellena de 5s.

Operaciones básicas: array

- Se pueden hacer operaciones directamente sobre el array (sin bucles for):

`arr + arr`

`arr * 10`

`arr - arr`

`1 / arr`

`arr ** 0.5`

`arr @ arr2` → Producto matricial

- Asignar valores mediante un filtro:

`a = np.arange(20)`

`print(a)` # [0,1, ..., 18, 19]

`a[a > 5] = 5` # Los mayores de 5 a 5, actúa sobre el valor.

`print(a)` [0,1,2,3,4,5,5,5 ..., 5]

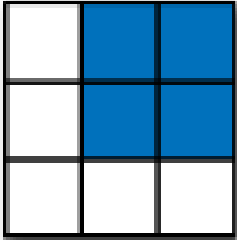
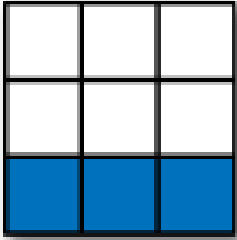
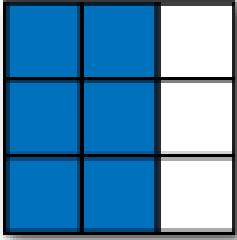
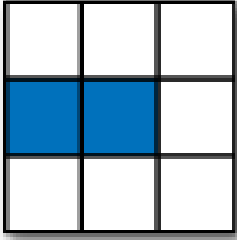
Operaciones básicas: array

- También están soportados los operadores relacionales a nivel de array: `>`, `>=`, `<`, `<=`, `==`, `!=`
 - El resultado lo muestra en otro array, indicando el resultado de la comparación.
- `arr = np.array([[1., 2., 3.], [4., 5., 6.]])`
- `arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])`
- `arr3 = arr > arr2`
- `array([[True, False, True], [False, True, False]])`

NumPy: array

- El acceso a los elementos con el []
 - `print(arr[0])` El primero.
 - Los corchetes también para modificar elementos:
 - `arr[1]=100`
- O con el método **item** indicando la posición:
 - `print(arr.item(1))` El segundo.
- **Slice:**
 - Los parámetros de slicing son los mismos que en Python → **[ini:fin-1:salto]**
 - La diferencia es que numpy son vistas de la matriz original. **NO se copian!**
 - Cuando queramos obtener una copia → **`arr[5:8].copy()`**
 - `print(arr[0:3])` Los 3 primeros
 - `arr[0:3] = 100` Los 3 primeros toman el valor 100.
- Utilizar doble: `arr[f][c]` cuando tenga el array más de una dimensión.

Slicing

	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>

Indexación booleana

- El objeto ndarray también soporta vectorización con matrices boolean.
- Podemos seleccionar filas o valores de un ndarray a partir de otro.
- Partimos de un array de nombres y una matriz de 7 x 4 aleatorios:
- In [98]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
- In [99]: data = np.random.randn(7, 4)
- In [100]: namesOut[100]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
- In [101]: data
- Out[101]: array([[0.0929, 0.2817, 0.769 , 1.2464],
[1.0072, -1.2962, 0.275 , 0.2289],
[1.3529, 0.8864, -2.0016, -0.3718],
[1.669 , -0.4386, -0.5397, 0.477],
[3.2489, -1.0212, -0.5771, 0.1241],
[0.3026, 0.5238, 0.0009, 1.3438],
[-0.7135, -0.8312, -2.3702, -1.8608]])

Indexación Booleana

- Disponemos de 7 nombres y 7 filas en data.
- Vamos a seleccionar las filas de data según el nombre sea 'Bob' en el array de nombres:
- `names == 'Bob'`
 - Devuelve: [True, False, False, True, ...]
- `data[names == 'Bob']`
- # La matriz boolean debe tener la misma longitud que el número de filas de la matriz.
- Nos devuelve la fila 0 y la 3 que son las posiciones que ocupa el nombre 'Bob' dentro del array de nombres.

Indexación Booleana

- Se puede seleccionar filtrando y utilizando slicing:
- Desde la columna 2 hasta el final:
 - `data[names == 'Bob', 2:]`
- Cogiendo la columna 3:
 - `data[names == 'Bob', 3]`
- Podemos seleccionar el resto utilizando el operador `!=` o `~` (útil para invertir una colección).
 - `data[~(names == 'Bob')]`

Indexación booleana

- `mask = (names == 'Bob') | (names == 'Will')`
- `data[mask]`
- Las palabras claves `and` y `or` de Python no funcionan aquí, hay que utilizar los símbolos: `&` |

Indexación elegante

- Consiste en seleccionar filas de la matriz a partir de matrices de enteros:
 - Podemos inicializar filas enteras:
 - In [117]: `arr = np.empty((8, 4))`
 - In [118]: `for i in range(8):`
 -: `arr[i] = i # Inicializa Toda la fila al valor de i`
 - In [119]: `arr`
 - Out[119]:
 - `array([[0., 0., 0., 0.],`
 - `[1., 1., 1., 1.],`
 - `[2., 2., 2., 2.],`
 - `[3., 3., 3., 3.],`
 - `[4., 4., 4., 4.],`
 - `[5., 5., 5., 5.],`
 - `[6., 6., 6., 6.],`
 - `[7., 7., 7., 7.]])`

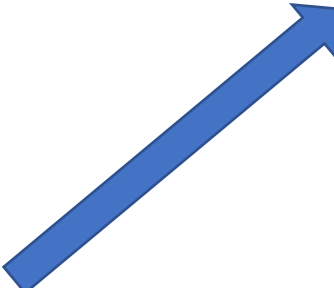
Indexación elegante

- También podemos seleccionar filas a partir de una matriz de enteros que representan las posiciones de las filas.
- In [120]: `arr[[4, 3, 0, 6]]` **# La posición de las índices marcan la posición.**
- Out[120]:
- `array([[4., 4., 4., 4.],`
- `[3., 3., 3., 3.],`
- `[0., 0., 0., 0.],`
- `[6., 6., 6., 6.]])`
- También se pueden utilizar índice negativos para seleccionar desde el final.

Indexación elegante

- Si pasamos una matriz de dos dimensiones, selecciona elementos de una matriz según los pares tomados de 2 en 2.
 - In [122]: `arr = np.arange(32).reshape((8, 4))`
 - In [123]: `arr`
 - Out[123]: `array([[0, 1, 2, 3],
[4, 5, 6, 7],
[8, 9, 10, 11],
[12, 13, 14, 15],
[16, 17, 18, 19],
[20, 21, 22, 23],
[24, 25, 26, 27],
[28, 29, 30, 31]])`
 - In [124]: `arr[[1, 5, 7, 2], [0, 3, 1, 2]]`
 - Out[124]: `array([4, 23, 29, 10])` **# Siempre devuelve 1 dimension.**

Selecciona las coordenadas:
(1,0)
(5,0)
(7,1)
(2,2)



Otro ejemplo

- Para obtener una forma rectangular:
- In [125]: `arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]`
- Out[125]:
- `array([[4, 7, 5, 6],`
- `[20, 23, 21, 22],`
- `[28, 31, 29, 30],`
- `[8, 11, 9, 10]])`

Trasposición

- Las matrices tiene el método `transpose` y el atributo especial `T`:
- In [126]: `arr = np.arange(15).reshape((3, 5))`
- In [127]: `arr`
- Out[127]:
- `array([[0, 1, 2, 3, 4],`
- `[5, 6, 7, 8, 9],`
- `[10, 11, 12, 13, 14]])`
- In [128]: **`arr.T`**
- Out[128]: `array([[0, 5, 10],`
- `[1, 6, 11],`
- `[2, 7, 12],`
- `[3, 8, 13],`
- `[4, 9, 14]])`

Trasposición

- Cuando tenemos más de dos dimensiones, el método **transpose** acepta una **tupla** con los número de eje para permutar los ejes:
- In [132]: `arr = np.arange(16).reshape((2, 2, 4))`
- In [133]: `arr`
- Out[133]:
- `array([[[0, 1, 2, 3],`
- `[4, 5, 6, 7]],`
- `[[8, 9, 10, 11],`
- `[12, 13, 14, 15]])`

- In [134]: `arr.transpose((1, 0, 2))` *# Se intercambian los dos primeros ejes.*
- Out[134]:
- `array([[[0, 1, 2, 3],`
- `[8, 9, 10, 11]],`
- `[[4, 5, 6, 7],`
- `[12, 13, 14, 15]])`

Operaciones básicas: array

- La función **swapaxes** intercambia ejes.

```
In [13]: arr = np.arange(16).reshape((2, 2, 4))
```

```
In [14]: arr
```

```
Out[14]: array([[[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7]],

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

```
In [15]: arr.swapaxes(1,2)
```

```
Out[15]: array([[[ 0,  4],
                  [ 1,  5],
                  [ 2,  6],
                  [ 3,  7]],

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

Operaciones básicas: array

- Redimensionar:
 - Genera un array de 3 x 5
 - `a = np.arange(15).reshape(3,5)`
 - Dentro de la llamada a reshape nos podemos encontrar un valor a -1 (solo en uno). En este caso numpy calcula la otra dimensión, el valor fijo que damos debe ser proporcional al valor original.
- Más operaciones:
 - `a = np.arange(10)`
 - `print(np.sqrt(a))`: raíz cuadrada
 - `print(np.exp(a))`: exponencial $\rightarrow e^x$

Operaciones básicas: array

- Se puede calcular el elemento mayor de cada array:
- Genera 2 arrays aleatorios y después genera otro array con los valores máximos:

```
x = np.random.randn(5) -- np.random.random(5)
```

```
x = x * 10
```

```
print(x)
```

```
y = np.random.randn(5)
```

```
y = y * 10
```

```
print(y)
```

```
print(np.maximum(x,y))
```

Se pueden generar arrays de un tamaño con valores Aleatorios dentro de un rango:

```
b1 = np.random.randint(1,30, 10)
```

Un array de 10 elementos entre 1 y 30

```
b2 = np.random.randint(1,30, (4,3))
```

Una tabla de 4 filas x 3 columnas con valores entre 1 y 30.

```
np.random.seed(123)
```

Establecer una semilla.

Op. Unarios

Function	Description
<code>abs</code> , <code>fabs</code>	Compute the absolute value element-wise for integer, floating point, or complex values. Use <code>fabs</code> as a faster alternative for non-complex-valued data
<code>sqrt</code>	Compute the square root of each element. Equivalent to <code>arr ** 0.5</code>
<code>square</code>	Compute the square of each element. Equivalent to <code>arr ** 2</code>
<code>exp</code>	Compute the exponent e^x of each element
<code>log</code> , <code>log10</code> , <code>log2</code> , <code>log1p</code>	Natural logarithm (base e), log base 10, log base 2, and $\log(1 + x)$, respectively
<code>sign</code>	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
<code>ceil</code>	Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element
<code>floor</code>	Compute the floor of each element, i.e. the largest integer less than or equal to each element
<code>rint</code>	Round elements to the nearest integer, preserving the dtype
<code>modf</code>	Return fractional and integral parts of array as separate array
<code>isnan</code>	Return boolean array indicating whether each value is NaN (Not a Number)
<code>isfinite</code> , <code>isinf</code>	Return boolean array indicating whether each element is finite (non-Inf, non-NaN) or infinite, respectively
<code>cos</code> , <code>cosh</code> , <code>sin</code> , <code>sinh</code> , <code>tan</code> , <code>tanh</code>	Regular and hyperbolic trigonometric functions
<code>arccos</code> , <code>arccosh</code> , <code>arcsin</code> , <code>arcsinh</code> , <code>arctan</code> , <code>arctanh</code>	Inverse trigonometric functions

Function	Description
<code>add</code>	Add corresponding elements in arrays
<code>subtract</code>	Subtract elements in second array from first array
<code>multiply</code>	Multiply array elements
<code>divide</code> , <code>floor_divide</code>	Divide or floor divide (truncating the remainder)
<code>power</code>	Raise elements in first array to powers indicated in second array
<code>maximum</code> , <code>fmax</code>	Element-wise maximum. <code>fmax</code> ignores NaN
<code>minimum</code> , <code>fmin</code>	Element-wise minimum. <code>fmin</code> ignores NaN
<code>mod</code>	Element-wise modulus (remainder of division)
<code>copysign</code>	Copy sign of values in second argument to values in first argument

Op. Binarios

`greater, greater_equal, less,
less_equal, equal, not_equal`

Realizar comparación de elementos, produciendo una matriz booleana (equivalente a infijo operadores `>`, `>=`, `<`, `<=`, `==`, `!=`)

`logical_and, logical_or,
logical_xor`

Calcular valor de verdad de elementos de la operación lógica (equivalente a infijo operadores `&` `|`, `^`)

Programación orientada a matrices

- Numpy permite la vectorización → reemplaza bucles con operaciones a nivel de matriz.
- Ejemplo: Evaluar la función $\sqrt{x^2 + y^2}$ en una cuadrícula de valores.
- Numpy proporciona la función `meshgrid` que recibe dos matrices de 1D y devuelve 2 matrices 2D.
- Generamos puntos:
 - `points = np.arange(-5, 5, 0.01)`
 - `xs, ys = np.meshgrid(points, points)`
- Evaluar la función:
 - `z = np.sqrt(xs ** 2 + ys ** 2)`

Representación gráfica de la función

- In [160]: `import matplotlib.pyplot as plt`
- In [161]: `plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()`
- Out[161]: `<matplotlib.colorbar.Colorbar at 0x7f8520cd1b38>`

- In [162]: `plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")`
- Out[162]: `Text(0.5,1,'Image plot of $\sqrt{x^2 + y^2}$ for a grid of values')`

Expresar lógica condicional

- La función `numpy.where` es una versión vectorizada de la expresión ternaria `x if condition else y`.
 - In [165]: `xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])`
 - In [166]: `yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])`
 - In [167]: `cond = np.array([True, False, True, True, False])`
- Ahora queremos tomar el valor de `xarr` cuando `cond` es `True` y en caso contrario tomaremos `yarr`.
- Para ello `numpy` dispone de la función `where`:
 - In [170]: `result = np.where(cond, xarr, yarr)`
 - In [171]: `result`
 - Out[171]: `array([1.1, 2.2, 1.3, 1.4, 2.5])`
- Genera una matriz en función de otra matriz.

Operaciones básicas: array

- Generación de matrices con **where**:
 - Con **where** se pueden utilizar **escalares** o **matrices**.
 - El primer parámetro debe ser una matriz. Los otros dos pueden ser escalares.
 - Si el elemento de la matriz es True, devuelve 2 y sino False.
 - Con este criterio se crea la nueva matriz.

```
>>> import numpy as np
>>> arr = np.random.randn(4,4)
>>> arr
array([[ 0.19190511, -1.62677673,  0.68801224, -0.91909337],
       [-0.7139788 , -0.61458499, -0.47701828,  1.29776566],
       [ 1.63463471,  0.03654563, -1.62133205,  0.02006779],
       [ 0.86293828, -1.59750495,  0.88577936, -1.88501118]])
>>> arrTF = arr > 0
>>> arrTF
array([[ True, False,  True, False],
       [False, False, False,  True],
       [ True,  True, False,  True],
       [ True, False,  True, False]])
>>> doses = np.where(arr > 0, 2, -2)
>>> doses
array([[ 2, -2,  2, -2],
       [-2, -2, -2,  2],
       [ 2,  2, -2,  2],
       [ 2, -2,  2, -2]])
```

Operaciones básicas: array

- Generación de matrices con **where**
- Se pueden combinar matrices y escalares:

```
>>> nueva = np.where(arr>0,2,arr)
>>> nueva
array([[ 2.          , -1.62677673,  2.          , -0.91909337],
       [-0.7139788 , -0.61458499, -0.47701828,  2.          ],
       [ 2.          ,  2.          , -1.62133205,  2.          ],
       [ 2.          , -1.59750495,  2.          , -1.88501118]])
```

Métodos matemáticos y estadísticos

- Un conjunto de funciones matemáticas que calculan estadísticas sobre una matriz completa o sobre los datos a lo largo de un eje son accesibles como métodos de la clase de matriz.
- In [177]: `arr = np.random.randn(5, 4)`
- In [178]: `arr`
- Out[178]:
 - `array([[2.1695, -0.1149, 2.0037, 0.0296],`
 - `[0.7953, 0.1181, -0.7485, 0.585],`
 - `[0.1527, -1.5657, -0.5625, -0.0327],`
 - `[-0.929 , -0.4826, -0.0363, 1.0954],`
 - `[0.9809, -0.5895, 1.5817, -0.5287]])`
- In [179]: `arr.mean()`
- Out[179]: `0.19607051119998253`
- In [180]: `np.mean(arr)`
- Out[180]: `0.19607051119998253`
- In [181]: `arr.sum()`
- Out[181]: `3.9214102239996507`

Se pueden aplicar sobre la matriz completa
O sobre un eje.

Si lo hacemos sobre la matriz completa puede
ser de dos formas:

Cómo método del array o como función de numpy
pasando como parámetro el array.

Métodos matemáticos y estadísticos

- Las funciones como `mean` y `sum` toman un argumento `axis` opcional que calcula la estadística sobre el eje dado, lo que da como resultado una matriz con una dimensión menos:
- In [182]: `arr.mean(axis=1)` ***# Calcular media por columnas***
- Out[182]: `array([1.022 , 0.1875, -0.502 , -0.0881, 0.3611])`
- In [183]: `arr.sum(axis=0)` ***# Calcular la suma por filas***
- Out[183]: `array([3.1693, -2.6345, 2.2381, 1.1486])`

Métodos matemáticos y estadísticos

- Otros métodos como `cumsum` y `cumprod` no se agregan, sino que producen una matriz de resultados intermedios:
 - In [184]: `arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])`
 - In [185]: `arr.cumsum()`
 - Out[185]: `array([0, 1, 3, 6, 10, 15, 21, 28])`
 - Estas funciones devuelven una matriz del mismo tamaño.
- Si la matriz es de varias dimensiones realiza cálculos parciales por eje indicado.

Métodos matemáticos y estadísticos

- Ejemplo: cumsum / cumprod

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

```
In [7]: arr
```

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

```
In [8]: arr.cumsum(axis=0)
```

```
Out[8]: array([[ 0,  1,  2],  
              [ 3,  5,  7],  
              [ 9, 12, 15]], dtype=int32)
```

Ha sumado de acumulativa por columnas

Para hacerlo por filas:

```
In [9]: arr.cumsum(axis=1)
```

```
Out[9]: array([[ 0,  1,  3],  
              [ 3,  7, 12],  
              [ 6, 13, 21]], dtype=int32)
```

Métodos matemáticos y estadísticos

- `b = np.array([3.4, 5, 6, 7.8, 9.07])`
- Operaciones estadísticas:
 - Desviación estándar:
`np.std(b)`
 - La varianza:
`np.var(b)`
 - Suma:
`np.sum(b)`
 - Media:
`np.mean(b)` o `np.average(b)`
 - Mínimo / Máximo de un array (a parte de la anteriores disponemos de:)
 - `np.amin(b)`
 - `np.amax(b)`
 - `np.ptp(b)` : Devuelve el rango de valores (máximo-mínimo) del array a lo largo de un eje dado.

Métodos matemáticos y estadísticos

- **Percentiles**

- `b = np.random.randint(1,30,20)`
- Array de 20 elementos entre 1 y 30
- Indicar el percentil con un valor de `[0, 100]`
- `np.percentile(b, 50)`
- A parte se puede pasar un array con los percentiles que queremos calcular.
- `np.percentile(b, np.array([0,25,50,75,100]))`

- **Correlación**

- Calcular la correlación entre dos variables. Dos variables A y B estarán correlacionadas si al disminuir los valores de A también disminuyen los del B y viceversa.

Ejemplo

```
import numpy as np
import matplotlib.pyplot as plt
```

```
np.random.seed(1)
```

```
# 1000 random integers between 0 and 50
```

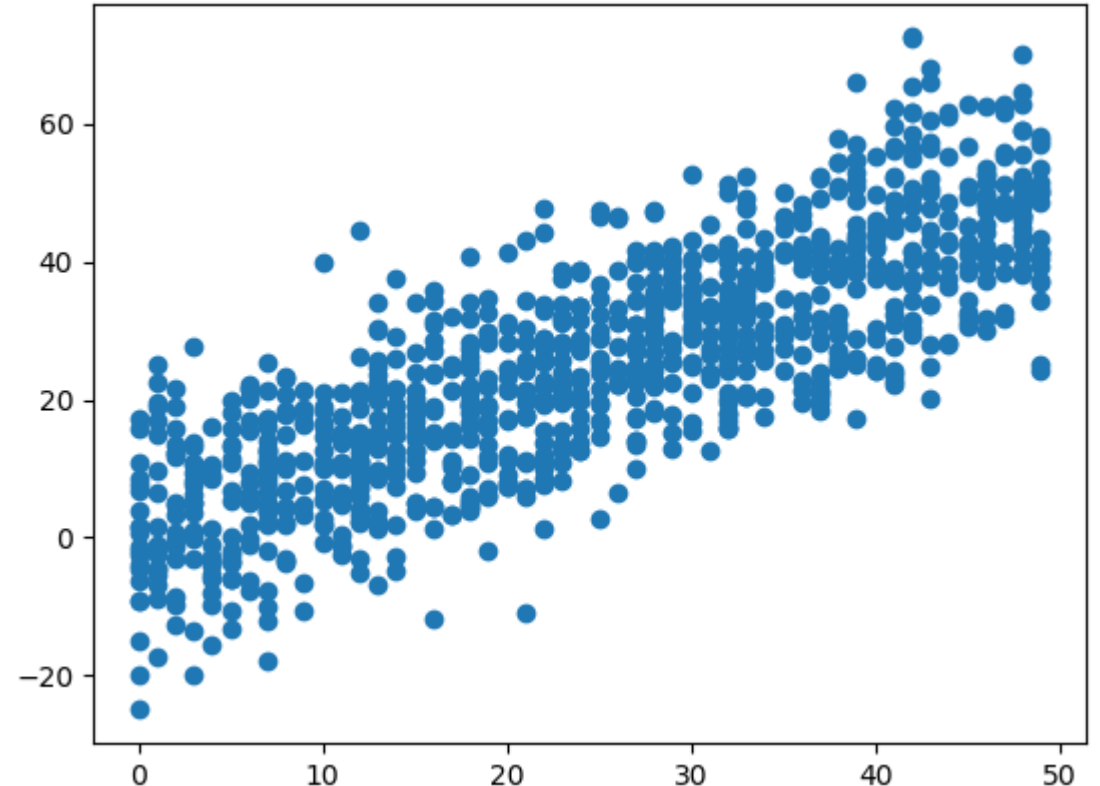
```
x = np.random.randint(0, 50, 1000)
```

```
# Positive Correlation with some noise
```

```
y = x + np.random.normal(0, 10, 1000)
```

```
plt.scatter(x,y)
```

```
c = np.corrcoef(x, y)
```



Histograma

- Numpy permite crear histogramas, hace un recuento de las veces que se repite cada valor en un array inicial.

```
import numpy as np
np.random.seed(1)
# 1000 numeros aleatorios entre 0 y 50
x = np.random.randint(0, 20, 1000)
index = np.arange(0,20)
v, i = np.histogram(x, index)
print(v) → Devuelve el recuento de cada valor
print(i) → Devuelve los índices de 0 a 20
```

- Podemos pasar los datos a un diccionario con:
 - `D = dict(zip(i,v))`

Method	Description
sum	Sum of all the elements in the array or along an axis. Zero-length arrays have sum 0.
mean	Arithmetic mean. Zero-length arrays have NaN mean.
std, var	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator n).
min, max	Minimum and maximum.
argmin, argmax	Indices of minimum and maximum elements, respectively.
cumsum	Cumulative sum of elements starting from 0
cumprod	Cumulative product of elements starting from 1

Métodos para matrices boolean

- Se valores boolean se relacionan 1 (True) y 0 (False). Se puede utilizar `sum` para contar valores True.
- Por ejemplo, calcular el número de valores > 0
 - In [190]: `arr = np.random.randn(100)`
 - In [191]: `(arr > 0).sum()`
 - Out[191]: 42
- Hay dos adicionales métodos **any** y **all**, especialmente útil para matrices booleanas.
 - **any**: prueba si uno o más valores en una matriz es True,
 - **all**: verifica si cada valor es True:

Ejemplo

- In [192]: `bools = np.array([False, False, True, False])`
- In [193]: `bools.any()` ***# Al menos uno es True***
- Out[193]: `True`
- In [194]: `bools.all()` ***# Todos son True***
- Out[194]: `False`
- Estos dos métodos también con matrices no Boolean. Considerando cualquier valor distinto de cero como True.

numpy: ordenar

- `a = np.random.randn(10)`
- `a *= 10`
- `a.sort()` # Ordena el array a.
- Lo Podemos invertir para ordenar desc:
 - `print(a[::-1])`
- También podemos ordenar por filas y por columnas (partiendo de una matriz)
 - `a.sort(0)` → por filas
 - `a.sort(1)` → por columnas
- Si utilizamos el método **`np.sort(a)`** nos retorna una copia ordenada.

- In [199]: arr = np.random.randn(5, 3)
- In [200]: arr
- Out[200]: array([[0.6033, 1.2636, -0.2555],
[-0.4457, 0.4684, -0.9616],
[-1.8245, 0.6254, 1.0229],
[1.1074, 0.0909, -0.3501],
[0.218 , -0.8948, -1.7415]])
- In [201]: arr.sort(1) **# Ordenación por filas**
- In [202]: arr
- Out[202]:
• array([[-0.2555, 0.6033, 1.2636],
[-0.9616, -0.4457, 0.4684],
[-1.8245, 0.6254, 1.0229],
[-0.3501, 0.0909, 1.1074],
[-1.7415, -0.8948, 0.218]])

Lógica única y de otro conjunto

- Numpy dispone de la función **np.unique** para quitar repetidos de un array.
 - `names = np.array(['Bob', 'Joe', 'Bob', 'Will', 'Joe'])`
 - `np.unique(names)`
 - Quita repetidos!
- Algo similar en Python haríamos: **`sorted(set(names))`**

Conjuntos

- La función **np.in1d** nos indica la pertenencia de los valores de una matriz a otra.
- Nos devuelve una matriz booleana.
 - `values = np.array([6,0,0,3,2,5,6])`
 - `arr2 = np.in1d(values, [2,3,6])`
- Con los valores True o False nos indican si los números de la matriz values se encuentran en la matriz [2,3,6]

Operaciones con conjuntos

- Otras operaciones del mismo estilo:
 - `np.intersect1d(x, y)` → Los elementos comunes
 - `np.union1d(x, y)` → La unión de conjuntos
 - `np.setdiff1d(x, y)` → La diferencia entre los conjuntos x e y. Los elementos de x que no están en y
 - `np.setxor1d(x, y)` → La diferencia simétrica: elementos que están en cualquiera de las matrices pero no en ambas.
 - Estas operaciones se corresponden con los operadores que tienen los conjuntos de Python: `&`, `|`, `-`, `^`

Numpy

Entrada / Salida

Entrada / Salida

- Los arrays se pueden almacenar y recuperar de archivos (**binarios**): **save** y **load**
- Si no ponemos la extensión (**.npy**) se la añade:

```
arr = np.arange(10)
print(arr)
np.save('array.bin.npy', arr)
arr2 = np.load('array.bin.npy')
print(arr2)
```

Entrada / Salida

- Los arrays se pueden almacenar y recuperar de archivos (de **texto**): **savetxt** y **loadtxt**
- Ejemplo:

```
np.savetxt('test1.txt', b, delimiter=',')  
bb = np.loadtxt('test1.txt', dtype=float)  
print(bb)
```
- Se pueden guardar varios arrays en el mismo archivo como si fuera una tupla (pero tienen que ser del mismo tamaño).

```
np.savetxt('varios.txt', (a,b,c))  
a2,b2,c2 = np.loadtxt('varios.txt')  
print(a2,b2,c2)
```

Entrada / Salida

- Numpy también puede cargar archivos con formato comprimido con el método load con extensión **npz**.
- curl -o http://www.ideo.columbia.edu/~rpa/argo_float_4901412.npz
argo_data = np.load('argo_float_4901412.npz')
- Para guardar arrays comprimidos:
a = np.arange(10)
b = np.arange(10)
np.savez_compressed('file.npz', a=a, b=b)
- Se pueden recuperar mediante las etiquetas: Similar a un diccionario (carga las matrices individuales de forma perezosa)
loaded = np.load('file.npz')
a_loaded = loaded['a']
b_loaded = loaded['b']

Numpy

Algebra Lineal

Algebra Lineal

- Permite realizar operaciones con matrices, descomposiciones, determinantes y otras operaciones con matrices cuadradas.
 - Multiplicación de matrices **@**
 - El producto de cada elementos *****
 - Producto escalar: **dot**
 - Producto interno: **inner**
 - Producto externo: **outer**

Algebra Lineal

- `x = np.array([1,8,0])`
- `y = np.array([5,-1,4])`
- `print(x * y)` # producto de cada elemento **$X_i * Y_i$**
- `print(x.dot(y))` # producto **Escalar, equivalente `np.dot(x,y)`**
- `print(inner(x,y))` # producto **Interno**: el primer array hace de fila y el segundo de col, y se suman los productos.

$$\begin{bmatrix} 1 & 8 & 0 \end{bmatrix} * \begin{bmatrix} 5 \\ -1 \\ 4 \end{bmatrix} = (1)(5) + (8)(-1) + (0)(4) = -3$$

Algebra Lineal

- **outer**

- Producto externo, se define como el producto de cada elemento de una fila de un vector X ($1 \times n$) por los elementos de una columna de un vector Y ($m \times 1$), da como resultado una matriz ($m \times n$)

- Ejemplo:

```
x = np.array([1,8,0])
```

```
y = np.array([5,-1,4])
```

```
print(np.outer(x,y))
```

$$\begin{bmatrix} 1 & 8 & 0 \end{bmatrix} * \begin{bmatrix} 5 \\ -1 \\ 4 \end{bmatrix} = \begin{bmatrix} (1)(5) & (1)(-1) & (1)(4) \\ (8)(5) & (8)(-1) & (8)(4) \\ (0)(5) & (0)(-1) & (0)(4) \end{bmatrix} = \begin{bmatrix} 5 & -1 & 4 \\ 40 & -8 & 32 \\ 0 & 0 & 0 \end{bmatrix}$$

Algebra Lineal

- Norma y determinante:
 - Para el **determinante** utilizamos la función **linalg.det** dentro de numpy:
 - `a = np.array([1,0,2,-1])`
 - `a = a.reshape(2,2)`
 - `det = np.linalg.det(a)`
 - Para la **norma**:
 - Disponemos de la función: **np.linalg.norm**
 - La norma se puede calcular de un vector o una matriz.
 - El primer parámetro será el array o la matriz y se puede pasar un segundo parámetro `ord` para indicar el tipo de norma a calcular:

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
'fro'	Frobenius norm	-
'nuc'	nuclear norm	-
inf	<code>max(sum(abs(x), axis=1))</code>	<code>max(abs(x))</code>
-inf	<code>min(sum(abs(x), axis=1))</code>	<code>min(abs(x))</code>
0	-	<code>sum(x != 0)</code>
1	<code>max(sum(abs(x), axis=0))</code>	as below
-1	<code>min(sum(abs(x), axis=0))</code>	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	-	<code>sum(abs(x)**ord)**(1./ord)</code>

Algebra Lineal

- Dentro del módulo **numpy.linalg** disponemos de:
- **T** para calcular la traspuesta
- **inv()** la inversa, se aplicará a una matriz cuadrada:
 - `numpy.linalg.inv(matriz)`

Funciones

Función	Descripción
diag	Regreso los elementos diagonales (o fuera de la diagonal) de una matriz cuadrada como una matriz 1D, o convierte una matriz 1D en una matriz cuadrada con ceros fuera de la diagonal
dot	Matriz multiplicación
trace	Calcular la suma de los elementos diagonales
det	Calcular el determinante de la matriz
eig	Calcular los autovalores y autovectores de una matriz cuadrada
inv	Calcular la inversa de una matriz cuadrada
pinv	Calcular la pseudoinversa de Moore-Penrose de una matriz
qr	Calcular la descomposición QR
svd	Calcular la descomposición en valor singular (SVD)

Funciones

Función	Descripción
<code>solve</code>	Resuelve el sistema lineal $Ax = b$ para x , donde A es una matriz cuadrada
<code>lstsq</code>	Calcular la solución de mínimos cuadrados a $Ax = b$

Operaciones básicas

- **Discretización:**
- `import numpy as np`
- `np.linspace(0.0, 1.0, 5)` # 5 intervalos entre 0.0 y 1.0
- `array([0. , 0.25, 0.5 , 0.75, 1.])`

Tipo Matrix

- El tipo está **deprecated**, utilizar ndarray
- Dentro de **numpy** también disponemos del tipo **np.matrix**
- Se puede crear a partir de listas de listas.
- `M = np.matrix([[3,9,-10], [1,-6,4], [10,-2,8]])`

Resolución de Ecuaciones

- Partimos de las ecuaciones:
- $3x + 9y - 10z = 24$
- $x - 6y + 4z = -4$
- $10x - 2y + 8z = 20$
- Creamos dos matrices, una con las ecuaciones y otra con los coeficientes.

Ejemplo

```
import numpy as np
```

Se crean las matrices:

```
a = np.matrix([[3,9,-10],[1,-6,4],[10,-2,8]])
```

```
print(a)
```

```
b = np.matrix([[24],[-4],[20]])
```

```
print(b)
```

Se calcula $X = \text{inv}(A) * B$

```
x = a**(-1)*b
```

```
print("Resultado:")
```

```
print(x)
```

Para verificar el resultado

Se debe cumplir que $X * A$ debe de dar B

```
x2 = a * x
```

```
print(x2)
```

```
"""
```

$$2x + 3y = 8$$

$$x - 2y = -10$$

```
"""
```

```
import numpy as np
```

```
A = np.matrix([[2, 3],[1, -2]])
```

```
b = np.matrix([[8],[-10]])
```

```
x = (A**-1)*b
```

```
# Resultado: [[-2], [4]]
```

Utilizar ndarray funciona y así no trabajamos con matrix que está descatalogado.

Se puede resolver con **`np.linalg.solve(a,b)`** Donde a es la matriz y b los coeficientes

Ejemplo con solve

PRUEBA CON SOLVE

```
In [31]: a = np.array([[1,2,1],[0,1,1],[1,0,1]])  
b = np.array([4,3,5])
```

```
In [33]: r = np.linalg.solve(a,b)
```

```
In [34]: r
```

```
Out[34]: array([ 1.5, -0.5,  3.5])
```

```
In [35]: a = np.array([[3,9,-10],[1,-6,4],[10,-2,8]])
```

```
In [36]: a
```

```
Out[36]: array([[ 3,  9, -10],  
                [ 1, -6,  4],  
                [10, -2,  8]])
```

```
In [37]: b = np.array([24,-4,20])
```

```
In [38]: b
```

```
Out[38]: array([24, -4, 20])
```

```
In [39]: resul = np.linalg.solve(a,b)
```

```
In [40]: resul
```

```
Out[40]: array([ 2.99029126,  0.40776699, -1.13592233])
```

Verificación del método

- Para comprobar que se puede resolver el sistema, la matriz tiene que ser invertible.
- Si no se puede invertir Python emite una excepción.
- Se puede comprobar si el determinante es $\neq 0$

```
if np.linalg.det(A) == 0:
```

```
    x = None
```

```
    print("No se puede resolver")
```

```
else:
```

```
    x = (A**-1)*b # Ojo podemos tener problemas con enteros  $< 0$ 
```

Numpy

Generación de números aleatorios

Numpy: Números aleatorios

- Disponemos del módulo: `numpy.random`
- Integra funciones para generar matrices de forma aleatoria utiliza funciones y distribuciones de probabilidad.
 - Distribución normal:
 - ejemplo = `np.random.normal(size=(4,4))`
 - Utilizar una semilla para inicializar:
 - `np.random.seed(1234)`
 - Se puede utilizar una semilla global:
 - `np.random.RandomState(1234)`

Numpy: Números aleatorios: numpy.random

Función	Descripción
<code>seed</code>	Sembrar el generador de números aleatorios
<code>permutation</code>	Devolver una permutación aleatoria de una secuencia, o devolver un rango permutado
<code>shuffle</code>	Al azar permutar una secuencia en el lugar
<code>rand</code>	Dibujar muestras de una distribución uniforme
<code>randint</code>	Dibujar números enteros aleatorios de un rango bajo a alto dado
<code>randn</code>	Dibujar muestras de una distribución normal con media 0 y desviación estándar 1 (interfaz similar a MATLAB)
<code>binomial</code>	Dibujar muestras de una distribución binomial

Numpy: Números aleatorios: `numpy.random`

<code>normal</code>	Dibujar muestras de una distribución normal (gaussiana)
---------------------	---

<code>beta</code>	Dibujar muestras de una distribución beta
-------------------	---

<code>gamma</code>	Dibujar muestras de una distribución gamma
--------------------	--

<code>uniform</code>	Dibujar muestras de una distribución uniforme $[0, 1)$
----------------------	--

```
x = np.random.randint(0, 50, 1000)
```

```
y = np.random.normal(0, 10, 1000)
```

Numpy

Operaciones Avanzadas

numpy: Operaciones avanzadas

- Se permite la definición de **arrays estructurados** definiendo campos con su tipo:
- Ejemplo: # S10, tamaño del string, si es mayor → trunca
tipo = [('nombre', 'S10'), ('altura', float), ('edad', int)]
valores = [('Arturo', 1.8, 44), ('Olga', 1.57, 55), ('Andrea', 1.65, 33)]

```
a = np.array(valores, dtype=tipo)
print(a)
print(np.sort(a, order='altura'))
```

numpy: Operaciones Avanzadas

- Se permite tener **arrays con máscaras** y estas se pueden utilizar para invalidar uno de los valores que no nos interesa.
- De tal forma que ese valor NO formará parte de la operación que vayamos a realizar.
- Necesario importar este módulo:
`import numpy.ma as ma`

numpy: Operaciones Avanzadas

- Ejemplo de **arrays con máscaras**:

```
c = np.array([1,1,1,500-7j,1])
```

- El valor se invalida marcando con un 1 en la máscara, los valores válidos se marcan con un 0:

```
m = ma.masked_array(c, mask=[0,0,0,1,0])
```

- Al imprimir el array el valor 3, aparece con ---
- Y no se tendrá en cuenta para las operaciones.

Más métodos

- **np.identity(n)**
 - Generar la **matriz identidad** de tamaño $n \times n$.
- Encontrar los **índices que cumplen cierta condición**:
 - `arr2 = np.argwhere(arr != 0)`
 - Genera un array con los índices de las posiciones que son distintas de 0.
- Obtener el **índice y el valor del máximo del array**:
 - `print(m.argmax())` → El índice del máximo
 - `print(m.ravel())` *# Pasa el array a una dimensión.*
 - `print(m.ravel()[m.argmax()])` *# Y lo indexa con el índice del máximo.*
- **Repetir tantas veces un array como indica n**:
 - `arr = np.arange(5)` *# [0,1,2,3,4]*
 - `print(arr)`
 - `m = np.tile(arr, 5)` *# [0,1,2,3,4,0,1,2,3,4,0,1,2,3,4 ...]*

Manipulación Avanzada de Arrays

- Reshaping de Arrays:
 - En muchos casos, puede convertir una matriz de una forma a otra sin copiar ningún dato. Para hacer esto, pase una tupla que indique la nueva forma al método de instancia: `reshape` de matriz.
 - In [20]: `arr = np.arange(8)`
 - In [21]: `arr`
 - Out[21]: `array([0, 1, 2, 3, 4, 5, 6, 7])`
 - In [22]: `arr.reshape((4, 2))`
 - Out[22]:
 - `array([[0, 1],`
 - `[2, 3],`
 - `[4, 5],`
 - `[6, 7]])`

Manipulación Avanzada de Arrays

- Una de las dimensiones de forma transferidas puede ser `-1`, en cuyo caso el valor utilizado para esa dimensión se deducirá de los datos:
- Ejemplo:
 - In [24]: `arr = np.arange(15)`
 - In [25]: `arr.reshape((5, -1))`
 - Out[25]:
 - `array([[0, 1, 2],`
 - `[3, 4, 5],`
 - `[6, 7, 8],`
 - `[9, 10, 11],`
 - `[12, 13, 14]])`

Manipulación Avanzada de Arrays

- Dado que el atributo `shape` de una matriz es una tupla, también se puede pasar a `reshape`:
 - In [26]: `other_arr = np.ones((3, 5))`
 - In [27]: `other_arr.shape`Out[27]: `(3, 5)`
 - In [28]: `arr.reshape(other_arr.shape)`
- Out[28]: `array([[0, 1, 2, 3, 4],`
- `[5, 6, 7, 8, 9],`
- `[10, 11, 12, 13, 14]])`

Manipulación Avanzada de Arrays

- El paso de dos dimensiones a una dimensión se realiza con el método `ravel()`:
 - In [29]: `arr = np.arange(15).reshape((5, 3))`
 - In [30]: `arr`
 - Out[30]: `array([[0, 1, 2],
[3, 4, 5],
[6, 7, 8],
[9, 10, 11],
[12, 13, 14]])`
 - In [31]: `arr.ravel()`
 - Out[31]: `array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])`

Manipulación Avanzada de Arrays

- El método `ravel` no produce una copia de los valores subyacentes si los valores del resultado eran contiguos en la matriz original.
- El método `flatten` se comporta como `ravel` excepto que siempre devuelve una copia de los datos:
- In [32]: `arr.flatten()`
- Out[32]: `array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])`

Orden C vs Fortran

- NumPy te da control y flexibilidad sobre el diseño de sus datos en la memoria. De forma predeterminada, las matrices NumPy se crean en orden de fila principal.
- Espacialmente, esto significa que si tiene una matriz de datos bidimensional, los elementos de cada fila de la matriz se almacenan en ubicaciones de memoria adyacentes.
- La alternativa a la colocación por filas es el orden por columna , lo que significa que los valores dentro de cada columna de datos se almacenan en ubicaciones de memoria adyacentes.

Orden C vs Fortran

- Por razones históricas, el orden principal de filas y columnas también se conocido como orden C y Fortran, respectivamente. En el lenguaje FORTRAN 77, todas las matrices son columnas principales.
- Los métodos reshape y ravel aceptan el argumento order que indica el orden para usar los datos en la matriz.
- Esto generalmente se establece en 'C' o 'F' en la mayoría de los casos (también hay opciones de uso menos común 'A' y 'K')

Ejemplo

- In [33]: `arr = np.arange(12).reshape((3,4))`
- In [34]: `arr`
- Out[34]: `array([[0, 1, 2, 3],
[4, 5, 6, 7],
[8, 9, 10, 11]])`
- In [35]: `arr.ravel()`
- Out[35]: `array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])`
- In [36]: `arr.ravel('F')`
- Out[36]: `array([0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11])`

Concatenación y división de matrices

- `numpy.concatenate` toma una secuencia (tupla, lista, etc.) de matrices y las une en orden a lo largo del eje indicado:
 - In [37]: `arr1 = np.array([[1, 2, 3], [4, 5, 6]])`
 - In [38]: `arr2 = np.array([[7, 8, 9], [10, 11, 12]])`
 - In [39]: `np.concatenate([arr1, arr2], axis=0)`
 - Out[39]:
 - `array([[1, 2, 3],`
 - `[4, 5, 6],`
 - `[7, 8, 9],`
 - `[10, 11, 12]])`
 - In [40]: `np.concatenate([arr1, arr2], axis=1)`
 - Out[40]: `array([[1, 2, 3, 7, 8, 9],`
 - `[4, 5, 6, 10, 11, 12]])`

Concatenación y división de matrices

- Numpy también dispone de las funciones **vstack** y **hstack** para concatenar las filas o las columnas respectivamente.
- La operación inversa **split** para dividir la matriz en varias matrices:
- Partimos de:
- In [43]: `arr = np.random.randn(5, 2)`
- In [44]: `arr`
- Out[44]:
- `array([[-0.2047, 0.4789],`
- `[-0.5194, -0.5557],`
- `[1.9658, 1.3934],`
- `[0.0929, 0.2817],`
- `[0.769 , 1.2464]])`

División de matrices: Ejemplo

- In [45]: `first, second, third = np.split(arr, [1, 3])`
- In [46]: `first`
- Out[46]: `array([[-0.2047, 0.4789]])`
- In [47]: `second`
- Out[47]: `array([[-0.5194, -0.5557],
[1.9658, 1.3934]])`
- In [48]: `third`
- Out[48]:
 - `array([[0.0929, 0.2817],
[0.769 , 1.2464]])`

El valor `[1, 3]` pasado a `np.split` indica los índices en los que dividir la matriz en partes.

Funciones

Función	Descripción
<code>concatenate</code>	Función más general, concatena la colección de matrices a lo largo de un eje
<code>vstack</code> , <code>row_stack</code>	Apilar matrices por filas (a lo largo del eje 0)
<code>hstack</code>	Apilar matrices en columnas (a lo largo del eje 1)
<code>column_stack</code>	Me gusta <code>hstack</code> , pero primero convierte matrices 1D en vectores de columna 2D
<code>dstack</code>	Apilar matrices "profundidad" en sentido (a lo largo del eje 2)
<code>split</code>	Matriz dividida en ubicaciones pasadas a lo largo de un eje particular
<code>hsplit</code> / <code>vsplit</code>	Conveniencia funciones para dividir en el eje 0 y 1, respectivamente

Objetos r_ y c_

- Sirven para realizar apilamientos de matrices: r_ (rows) / c_ (cols)

```
>>> import numpy as np
>>> arr = np.arange(6)
>>> arr1 = arr.reshape((3,2))
>>> arr1
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> arr2 = np.random.randn(3,2)
>>> arr2
array([[ 0.24770974,  0.06779006],
       [-1.02782512,  0.15755493],
       [ 0.71002583,  0.52731554]])
>>> np.r_[arr1, arr2]
array([[ 0.,          1.,          ],
       [ 2.,          3.,          ],
       [ 4.,          5.,          ],
       [ 0.24770974,  0.06779006],
       [-1.02782512,  0.15755493],
       [ 0.71002583,  0.52731554]])
>>> np.c_[arr1, arr2]
array([[ 0.,          1.,          ,  0.24770974,  0.06779006],
       [ 2.,          3.,          , -1.02782512,  0.15755493],
       [ 4.,          5.,          ,  0.71002583,  0.52731554]])
```

Se puede combinar en expresiones mas complejas:
np.c_[np.r_[arr1, arr2], arr]

Tile y Repeat

- Estas dos funciones nos sirven para repetir o replicar arrays.
 - In [55]: `arr = np.arange(3)`
 - In [56]: `arr`
 - Out[56]:
 - `array([0, 1, 2])`
 - In [57]: `arr.repeat(3)`
 - Out[57]: `array([0, 0, 0, 1, 1, 1, 2, 2, 2])`
 - In [58]: `arr.repeat([2, 3, 4])`
 - Out[58]: `array([0, 0, 1, 1, 1, 2, 2, 2, 2])`

Repeat

- Las matrices multidimensionales pueden tener sus elementos repetidos a lo largo de un eje particular.
- In [59]: `arr = np.random.randn(2, 2)`
- In [60]: `arr`
- Out[60]:
 - `array([[-2.0016, -0.3718],`
 - `[1.669 , -0.4386]])`
- In [61]: `arr.repeat(2, axis=0)`
- Out[61]:
 - `array([[-2.0016, -0.3718],`
 - `[-2.0016, -0.3718],`
 - `[1.669 , -0.4386],`
 - `[1.669 , -0.4386]])`

Tile

- tile, por otro lado, es un atajo para apilar copias de una matriz a lo largo de un eje.
- In [64]: arr
- Out[64]:
- array([[-2.0016, -0.3718],
[1.669 , -0.4386]])

- In [65]: np.tile(arr, 2) **# Replica 2 veces!**
- Out[65]:
- array([[-2.0016, -0.3718, -2.0016, -0.3718],
[1.669 , -0.4386, 1.669 , -0.4386]])

- **# Con una sola dimension:**
- arr = np.arange(3) → array([0,1,2])
- np.tile(arr, 3)
- array([0,1,2,0,1,2,0,1,2])

Otra forma de indexar: take / put

- In [69]: `arr = np.arange(10) * 100` # **array([0,100,200,300,...,900])**
- In [70]: `inds = [7, 1, 2, 6]`
- In [71]: `arr[inds]` # **Indica los índices que seleccionamos.**
- Out[71]: `array([700, 100, 200, 600])`
- El array de numpy dispone de los métodos: take y put

Ejemplo

- **# arr = array([0,100,200,300,...,900])**
- In [70]: inds = [7, 1, 2, 6]
- In [72]: arr.**take**(inds)
- Out[72]: array([700, 100, 200, 600])
- In [73]: arr.**put**(inds, 42)
- In [74]: arr
- Out[74]: array([0, **42**, **42**, 300, 400, 500, **42**, **42**, 800, 900])
- In [75]: arr.put(inds, [40, 41, 42, 43])
- In [76]: arr
- Out[76]: array([0, **41**, **42**, 300, 400, 500, **43**, **40**, 800, 900])

take / put con 2D

- Con el método take se puede utilizar el parámetro axis.
- Tenemos una matriz de 2D, y un array con los índices. Si en el parámetro axis indicamos un 1.
- El método take devuelve las columnas indicadas en los índices.
- En cambio put no acepta el argumento axis, indexa una matriz aplanada (de 1D).

Ejemplo

- In [77]: inds = [2, 0, 2, 1]
- In [78]: arr = np.random.randn(2, 4)
- In [79]: arr
- Out[79]:
- array([[-0.5397, 0.477 , 3.2489, -1.0212],
- [-0.5771, 0.1241, 0.3026, 0.5238]])

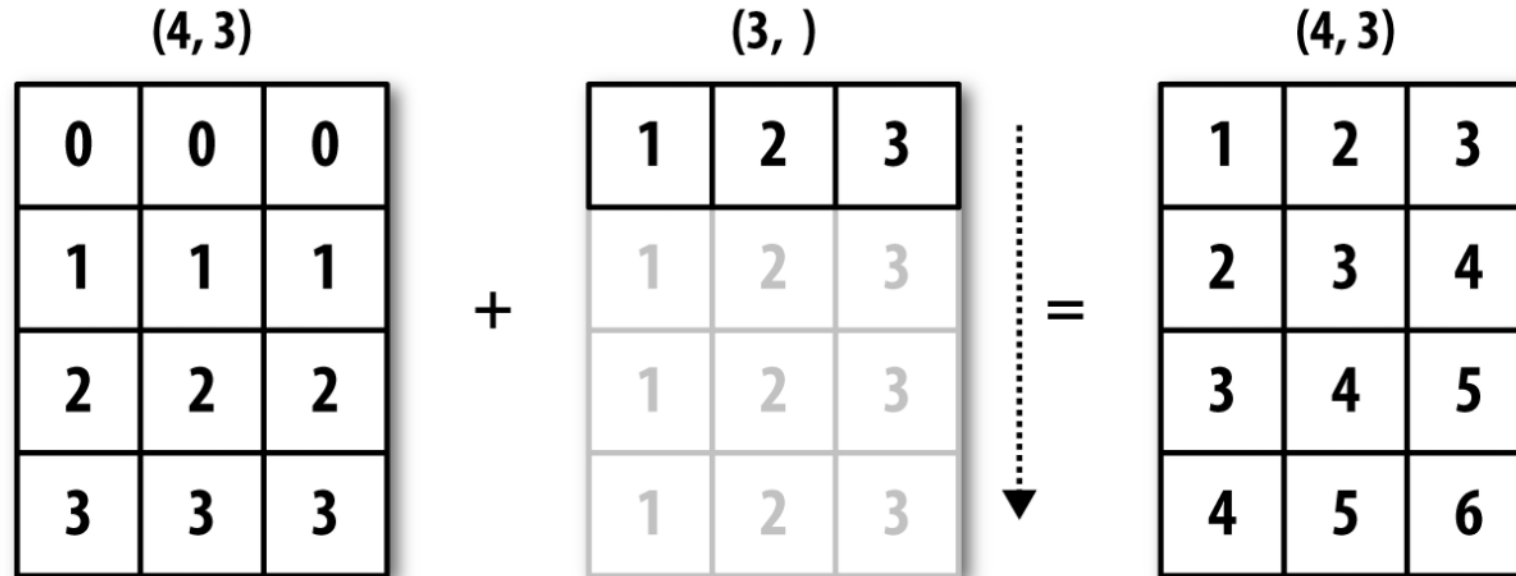
- In [80]: arr.take(inds, axis=1)
- Out[80]:
- array([[3.2489, -0.5397, 3.2489, 0.477],
- [0.3026, -0.5771, 0.3026, 0.1241]])

Broadcasting

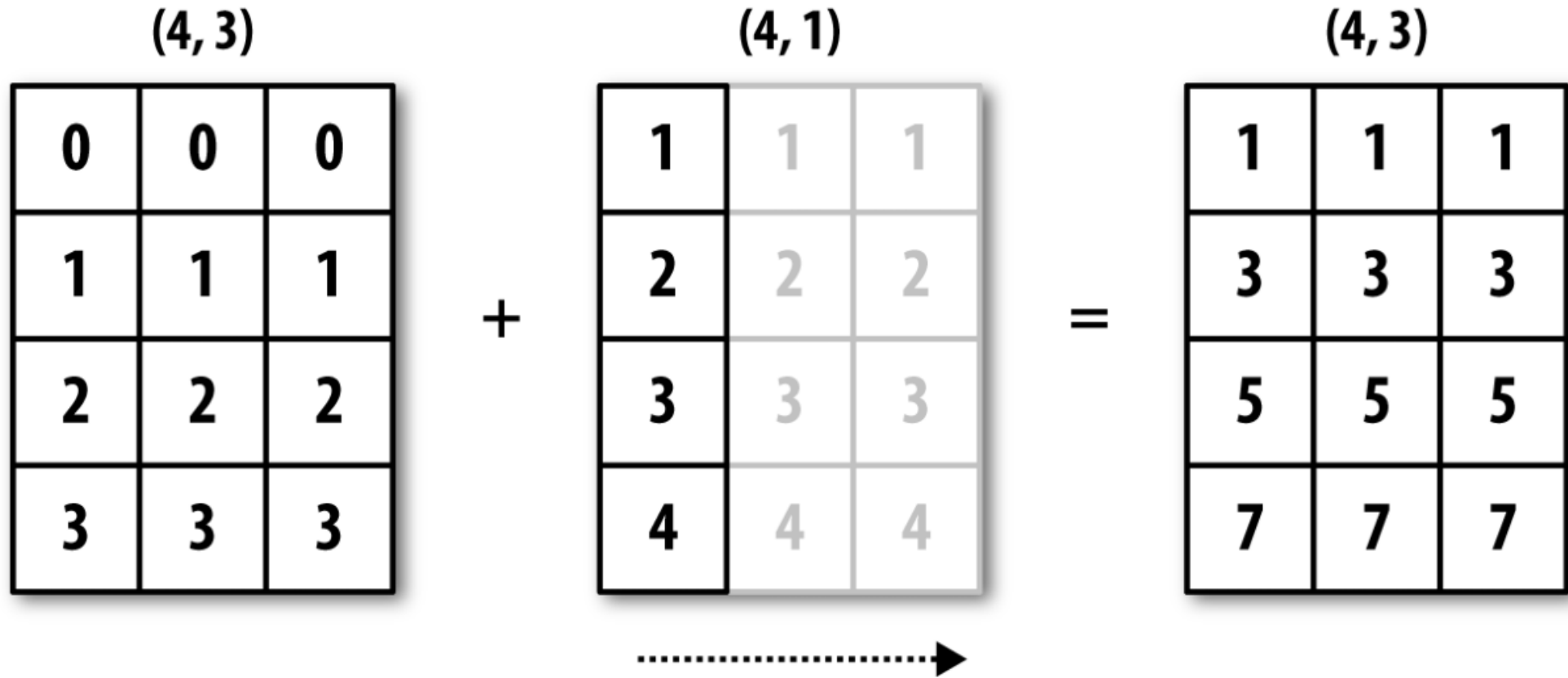
- La radiodifusión describe cómo funciona la aritmética entre matrices de diferentes formas.
- Se puede aplicar de forma sencilla, calculando el producto de un array por un escalar:
 - In [81]: `arr = np.arange(5)`
 - In [82]: `arr`
 - Out[82]: `array([0, 1, 2, 3, 4])`
 - In [83]: `arr * 4`
 - Out[83]: `array([0, 4, 8, 12, 16])`

Broadcasting (reglas)

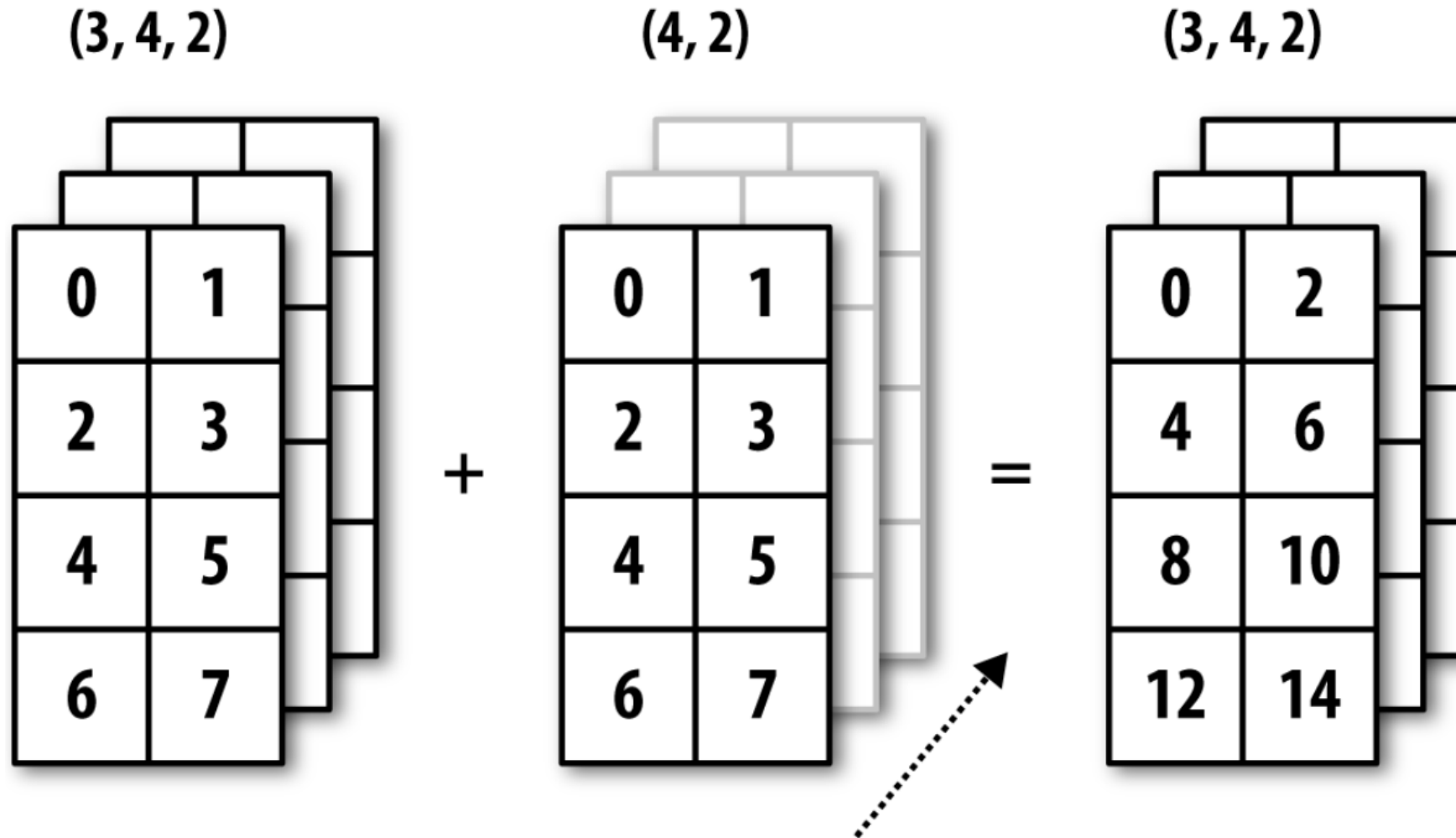
- Dos matrices son compatibles para la transmisión si para cada dimensión final (es decir, comenzando desde el final) las longitudes de los ejes coinciden o si cualquiera de las longitudes es 1. La transmisión se realiza luego sobre las dimensiones faltantes o longitud 1.



Broadcasting (reglas)



Broadcasting (reglas)



Configuración de valores de matriz por difusión

- La misma regla de difusión que rige las operaciones aritméticas también se aplica a la configuración de valores mediante la indexación de matrices.
- En un caso simple, podemos hacer cosas como:
 - In [109]: `arr = np.zeros((4, 3))`
 - In [110]: `arr[:] = 5`
 - In [111]: `arr`
 - Out[111]:
 - `array([[5., 5., 5.],`
 - `[5., 5., 5.],`
 - `[5., 5., 5.],`
 - `[5., 5., 5.]])`

Ejemplo (1 de 2)

- In [112]: `col = np.array([1.28, -0.42, 0.44, 1.6])`
- In [113]: `arr[:, :] = col[:, np.newaxis]`
- In [114]: `arr`
- Out[114]:
- `array([[1.28, 1.28, 1.28],`
- `[-0.42, -0.42, -0.42],`
- `[0.44, 0.44, 0.44],`
- `[1.6 , 1.6 , 1.6]])`

Ejemplo (2 de 2)

- In [115]: `arr[:2] = [[-1.37], [0.509]]`
- In [116]: `arr`
- Out[116]:
- `array([[-1.37 , -1.37 , -1.37],`
- `[0.509, 0.509, 0.509],`
- `[0.44 , 0.44 , 0.44],`
- `[1.6 , 1.6 , 1.6]])`