



Módulo 1 Numpy

Curso de Python Científico

César Husillos Rodríguez

IAA-CSIC

Septiembre de 2014

Introducción

¿Qué es NUMPY?

NUMPY es el **paquete** fundamental para el trabajo de computación científica con PYTHON. Contiene

- | | |
|------------------|-------------|
| 1 Tipos de datos | 3 Funciones |
| 2 Clases | 4 Módulos |

que posibilitan la creación y manejo de arrays n-dimensionales.

- El tipo de dato más importante es el array (o ndarray, de n-dimensional array).

Introducción

Ventajas

- 1 **Multiplataforma.**
- 2 **Potente** conjunto de librerías.
Puede clonar la funcionalidad de cualquier otro paquete de cálculo numérico (MATEMÁTICA, MATLAB)
- 3 Perfecta **integración con el Core** de PYTHON.
Interactúa con los tipos de datos y estructuras propios del lenguaje.

Se pueden crear arrays de NUMPY a partir de estructuras de PYTHON.

- 4 **Gratuito.**

Introducción

¿Qué vamos a aprender en este Módulo?

- 1 **Fundamentos sobre NUMPY:** creación de arrays, indexación, operaciones (aritméticas, relacionales y lógicas), cambio de tipo de datos y valores especiales.
- 2 **Aplicaciones de NUMPY** al tratamiento de datos científicos: selección, ordenación, ajustes, interpolaciones, estadística, polinomios, funciones matemáticas, arrays con máscaras, ...

Arrays

Definición

Un array es...

- Un tipo de dato compuesto a partir de tipos de datos sencillos.
- Están ordenados según una secuencia definida.

¿Como en una lista o una tupla?

- Sí en cuanto al requisito de ordenación.

- No en cuanto al contenido, porque sólo admite **un tipo de dato** por array.

Arrays

Algunos tipos de datos

Sn	Cadena de texto (<code>string</code>) de n-caracteres
bool	Booleano (True o False). Se almacena como 1 bit
int	Entero (int32 o int64, dependiendo de la plataforma)
int8	Byte (-128 a 127)
int16	Entero (-32768 a 32767)
int32	Entero (-2.147.483.648 a 2.147.483.647)
int64	Entero (-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807)
uint8	Entero sin signo (0 a 255)
uint16	Entero sin signo (0 a 65535)
uint32	Entero sin signo (0 a 4.294.967.295)
uint64	Entero sin signo (0 a 18.446.744.073.709.551.615)
float	Atajo para float64
float32	Decimal en precisión simple.
float64	Decimal en doble precisión.
complex	Atajo a complex128
complex64	Número complejo, parte entera e imaginaria con float32
complex128	Número complejo, parte entera e imaginaria con float64

Los tipos se referencian como cadena (`'int'`) o como constante numpy (`numpy.int`).

Arrays

Propiedades

Un array es un objeto y tiene propiedades (y métodos):

Propiedad	Descripción
<code>ndarray.shape</code>	Tupla con las dimensiones.
<code>ndarray.ndim</code>	Número de dimensiones.
<code>ndarray.size</code>	Número de elementos.
<code>ndarray.itemsize</code>	Tamaño de uno de los elementos en bytes.
<code>ndarray.nbytes</code>	Tamaño total ocupado por los elementos.
<code>ndarray.dtype</code>	Tipo de dato de los elementos.
<code>ndarray.real</code>	Parte real.
<code>ndarray.imag</code>	Parte imaginaria.

Arrays

Propiedades: ejemplo

```
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.shape
(2, 3)
>>> a.ndim
2
>>> a.size
6
>>> a.itemsize
8
```

```
>>> a.nbytes
48
>>> a.dtype
dtype('int64')
>>> a.real
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.imag
array([[0, 0, 0],
       [0, 0, 0]])
```


Manejo básico

Índice

- 1 Creación.
- 2 Indexación.
- 3 Operaciones aritméticas (reglas de broadcasting), relacionales y lógicas.
- 4 Cambio de tipo (casting).
- 5 Valores especiales (nan e inf).

Manejo básico

Creación de arrays

En principio, veremos 4 formas:

- 1 A partir de secuencias (listas o tuplas) de PYTHON.
- 2 Haciendo uso de funciones propias de NUMPY.
- 3 Lectura de datos desde fichero.
- 4 Copiando otro array.

Creación de arrays

1. Uso tipos de dato nativos de PYTHON

```
numpy.array(secuencia, tipo_dato)
```

```
>>> import numpy
>>> numpy.array([1, '5', 4.3, 1+3j]) # “casting” implícito
array(['1', '5', '4.3', '(1+3j)'], dtype='|S6')
>>> tupla = (3, 5, 7.7)
>>> a2 = numpy.array(tupla)
>>> a2
array([ 3. ,  5. ,  7.7])
>>> a3 = numpy.array([])
>>> a3
array([], dtype=float64)
>>> a4 = numpy.array(['linea1', 'linea2', 33], dtype='|S3')
array(['lin', 'lin', '33'],
      dtype='|S3')
>>> # “casting” explícito
```

Creación de arrays

1. Uso tipos de dato nativos de PYTHON

Usando una lista de listas.

Arrays n-dimensionales

```
>>>a = numpy.array([[1, 2, 3, 4],[5, 6, 7, 8], [9, 10, 11, 12]], \
dtype=numpy.int)
>>> a.shape
(3, 4)
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Creación de arrays

2. Mediante el uso de funciones de NUMPY

```
numpy.arange([start], stop[, step], dtype=None)
```

Equivalente a la función “range(start, stop, step)” de PYTHON.

A tener en cuenta...

1. El “step” **puede ser decimal** (novedad!!).
2. El extremo final del intervalo **no** se incluye.

```
>>> numpy.arange(5, 6, 0.1)
array([ 5. ,  5.1,  5.2,  5.3,  5.4,  5.5,
        5.6,  5.7,  5.8,  5.9])
```

Creación de arrays

2. Mediante el uso de funciones de NUMPY

```
numpy.linspace(start, stop, num=50,  
               endpoint=True, retstep=False)
```

Devuelve un array en el que se ha dividido el intervalo [start, stop] (**endpoint=True, por defecto**) en “num” fragmentos.

```
>>> numpy.linspace(5, 6, 5)  
array([ 5.   ,  5.25,  5.5   ,  5.75,  6.   ])  
>>> numpy.linspace(5, 6, 5, False, True)  
(array([ 5.   ,  5.2,  5.4,  5.6,  5.8]), 0.2)
```

Creación de arrays

2. Mediante el uso de funciones de NUMPY

```
numpy.ones(shape, dtype=None)
```

- `shape`, es la forma del array de salida (entero o lista/tupla). Si le pasamos una lista o tupla, crea un array n-dimensional con la forma (`shape`) dada por la lista/tupla.
- `dtype`, cualquiera de los tipos de datos de NUMPY.

```
>>> numpy.ones(4, dtype=numpy.complex128)
array([ 1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j])
>>> numpy.ones([2,3], dtype='int')
array([[1, 1, 1],
       [1, 1, 1]])
```

Creación de arrays

2. Mediante el uso de funciones de NUMPY

```
numpy.zeros(shape, dtype=float)
```

Exactamente el mismo comportamiento que `numpy.ones`.

```
>>> numpy.zeros((2,3,4))  
array([[[ 0.,  0.,  0.,  0.],  
        [ 0.,  0.,  0.,  0.],  
        [ 0.,  0.,  0.,  0.]],  
       [[ 0.,  0.,  0.,  0.],  
        [ 0.,  0.,  0.,  0.],  
        [ 0.,  0.,  0.,  0.]])
```


Creación de array

3. Lectura de ficheros

La función de lectura y sus parámetros dependen del formato del fichero.

Supongamos el fichero “datos.csv” con el formato siguiente:

```
line 1 -> objID,RAJ2000,e_RAJ2000,DEJ2000,e_DEJ2000,upmag,e_upmag,gpmag,e_gpmag,rpmag,e_rpmag
line 2 -> 1237657610717364296,138.692294,0.002,46.253899,0.002,18.049,0.015,16.904,0.033,16.
...
```

- 1 La primera línea contiene el nombre de los campos.
- 2 Las siguientes, los valores de cada objeto, separados por “,”.

Creación de arrays

3. Lectura de ficheros

```
numpy.loadtxt(fname, dtype=numpy.float, comments="#",  
              delimiter=None, converters=None, skiprows=0,  
              usecols=None, unpack=False, ndmin=0)
```

```
>>> array = numpy.loadtxt('datos.csv', \  
delimiter=',', skiprows=1)  
>>> array.dtype  
dtype('float64')  
>>> array.size  
189  
>>> array.shape  
(9, 21)
```

Creación de arrays

4. Mediante copia de otro array

Para copiar un array, basta con asignarlo a otra variable.

```
>>> a = numpy.arange(3)
>>> b = a  # asignacion peligrosa!!
>>> b[0] = -3
>>> b
array([-3,  1,  2])
>>> a
array([-3,  1,  2])
```

ATENCIÓN: Copia y original **comparten memoria**.

Este tipo de copia se denomina **copia por referencia**.

Creación de arrays

4. Mediante copia de otro array

Hay alternativas para la copia de arrays de forma que uno y otro sean objetos diferentes:

- Crear uno a partir de una operación con el otro.
- Usar la función `copy` de NUMPY.

Este tipo de copia se denomina **copia por valor**.

Creación de arrays

4. Mediante copia de otro array

Mediante operación ...

```
>>> a = numpy.arange(3)
>>> a
array([0, 1, 2])
>>> # Operacion aritm.
>>> b = a + 0
>>> b[0] = -3
>>> b
array([-3, 1, 2])
>>> a
array([0, 1, 2])
```

Método/Función copy...

```
>>> a = numpy.arange(3)
>>> # metodo
>>> b = a.copy()
>>> # funcion
>>> b = numpy.copy(a)
>>> b[0] = -3
>>> b
array([-3, 1, 2])
>>> a
array([0, 1, 2])
```

Manejo básico de arrays

Indexación

Se refiere a la selección de elementos concretos del array.

A tener en cuenta...

- Se accede a un elemento del array dando su **posición** en el array, mediante un índice ENTERO entre corchetes (' [] ')

`nombre_array[posicion]`

- El **primer índice** es el 0 (como en C/C++).
- Si el **índice es mayor** que el número de elementos de array, **lanzaré una excepción** (IndexError).

Indexación

Posibilidades

Tipo de selección

Un sólo elemento

Varios elementos consecutivos

Elementos en orden cualesquiera

(Novedad respecto a PYTHON Core.)

Sintaxis

`array[posicion]`

`array[inicio:fin]`

`array[[p1, p2, ..., pn]]`

donde `[p1, ..., pn]` es una lista o array.

Recordatorio

Los índices pueden tomar valores negativos. Al igual que en las secuencias de PYTHON cuentan las posiciones desde el final del array.

Indexación

Posibilidades

```
>>> a = numpy.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[1], a[-1]
(1, 9)
>>> a[3:-4]
array([3, 4, 5])
>>> a[0:9:2]
array([0, 2, 4, 6, 8])
>>> a[[3,5,9]]
array([3, 5, 9])
```

NOTA: El extraer varios elementos de un array **¡genera otro array!**.

Complicando un poco...

Arrays N-dimensionales

Arrays n-dimensionales

Creación

- Se pueden crear desde cero como hemos visto.
 - Usando la función `array` y pasándole una lista/tupla de listas/tuplas.
 - Usando funciones NUMPY que tengan el parámetro `shape`.
 - Leyendo desde fichero.
 - Copiando/extrayendo submatriz de otro array.
- O se puede modificar un array existente, haciendo uso de la **propiedad** `shape` o del **método** `reshape`.

Arrays n-dimensionales

Creación

Creación ad-hoc

```
>>> numpy.ones((2, 3))  
array([[ 1.,  1.,  1.],  
       [ 1.,  1.,  1.]])
```

Uso de la propiedad shape

```
>>> a = numpy.ones(10)  
>>> a.shape  
(10,)  
>>> a.shape = [2,5]  
>>> a  
array([[ 1.,  1.,  1.,  1.,  1.],  
       [ 1.,  1.,  1.,  1.,  1.]])
```

Arrays n-dimensionales

Creación

Uso de la método reshape

```
>>> a = numpy.ones(10)
>>> a.shape
(10,)
>>> b = a.reshape((2,5))  # lista o tupla
>>> a.shape
(10,)
>>> b.shape
(2, 5)
```

Atención a las dimensiones finales

```
>>> c = a.reshape((3,4))
ValueError: total size of new array must be unchanged
```

Arrays n-dimensionales

Indexado

- Se hace referencia a cada elemento del array con tantos índices, separados por comas, como dimensiones tiene el array.
- La combinación de índices va entre corchetes tras el nombre del array.

```
nombreArray[indexDim1, indexDim2, ...,  
             indexDimN]
```

Arrays n-dimensionales

Indexado

El **operador ':'** sustituye a todo el rango de índices posibles en la dimensión en la que aparece.

```
>>> a = numpy.linspace(0, 1, 5)
>>> a
array([ 0.   ,  0.25,  0.5  ,  0.75,  1.   ])
>>> a[:]
array([ 0.   ,  0.25,  0.5  ,  0.75,  1.   ])
>>> b = numpy.arange(4).reshape((2,2))
>>> b
array([[0, 1],
       [2, 3]])
>>> b[:,1]
array([1, 3])
>>> b[0, :]
array([0, 1])
```

Ejemplo

Indexado

```
>>> a = numpy.arange(24, \
dtype='int')
>>> a = a.reshape((2, 3, 4))
>>> a
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]]])
```

```
>>> a[0]    # equivalente a:
            #      a[0,:,:]
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a[0,0]  # equivalente a:
            #      a[0, 0, :]
array([0, 1, 2, 3])
>>> a[0,0,0]
0
```

Pregunta...

¿Que resultado obtendría si escribo `a[1,1:3,:2]`?
(Tómese su tiempo y escriba la respuesta en un papel)

Indexado

Arrays n-dimensionales

Respuesta

```
>>> a
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]]])

>>> a[1,1:3,:2]
array([[16, 17],
       [20, 21]])
```


Otras alternativas para cambiar la "forma" de un array

Algo parecido a lo que hacíamos con las listas de PYTHON:

Función	Descripción
<code>numpy.insert(arr, obj, values, axis=None)</code>	Inserta en las posiciones dadas por "obj" del eje "axis", los valores "values" (escalar o secuencia).
<code>numpy.append(arr, values, axis=None)</code>	Agrega al final del array "arr" en el eje "axis" el valor o valores dados por "values" (escalar o secuencia).
<code>numpy.delete(arr, obj, axis=None)</code>	Devuelve un array en el que se han borrado los elementos dados por los índices "obj" del eje "axis".

```
>>> a = numpy.arange(3)
>>> print numpy.insert(a, (0, -1), -1)
[-1  0  1  -1  2]
>>> print numpy.append(a, -3)
[ 0  1  2 -3]
>>> print numpy.delete(a, 0)
[1  2]
```

Otras alternativas para cambiar la "forma" de un array

Método

`array.flatten(order='C')`

Método que permite cambiar de un array n-dimensional a otro 1D.

```
>>> a = numpy.arange(6).reshape((3, 2))
>>> print a
[[0 1]
 [2 3]
 [4 5]]
>>> print a.flatten(order="C")
[0 1 2 3 4 5]
>>> print a.flatten(order="F")
[0 2 4 1 3 5]
```

Otras alternativas para cambiar la "forma" de un array

Función

```
numpy.ravel(array, order='C')
```

Función que permite cambiar de un array n-dimensional a otro 1D.

```
>>> a = numpy.arange(10).reshape((5,2))
>>> print a
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
>>> print numpy.ravel(a, order="C")
[0 1 2 3 4 5 6 7 8 9]
>>> print numpy.ravel(a, order="F")
[0 2 4 6 8 1 3 5 7 9]
```

Otras alternativas para cambiar la "forma" de un array

Función de concatenación de arrays.

Función

```
numpy.concatenate((a1, a2, ...), axis=0)
```

con

- “(a1, a2, ...)” una secuencia de arrays. Su “shape” debe coincidir, a excepción de la dimensión dada por “axis”.
- “axis” es la dimensión donde se van a unir los arrays.

Otras alternativas de cambiar la "forma" de un array

```
>>> a = numpy.array([[1, 2], [3, 4]])
>>> print a.shape
(2, 2)
>>> b = numpy.array([[5, 6]]) # <--- ATENCION
>>> print b.shape
(1, 2)
>>> print numpy.concatenate((a, b), axis=0)
[[1 2]
 [3 4]
 [5 6]]
>>> print b.T.shape
(2, 1)
>>> print numpy.concatenate((a, b.T), axis=1)
[[1 2 5]
 [3 4 6]]
```

NOTA: "b.T" retorna la **traspuesta** del array "b".

Otras alternativas para cambiar la "forma" de un array

Matiz importante: los arrays creados según las instrucciones

- `a = numpy.array([1, 2, 3])`
- `b = numpy.array([[1, 2, 3]])`

son muy diferentes. ¿Por qué?

Porque las **dimensiones** de uno y otro array son **diferentes**.

A la hora de concatenar, el número de dimensiones de los arrays deben ser las mismas.

Otras alternativas para cambiar la "forma" de un array

```
>>> import numpy
>>> a.shape
(3,) # array UNI-dimensional
>>> b.shape
(1, 3) # array BI-dimensional
>>> c = numpy.zeros((3,3))
>>> numpy.concatenate((a,c))
```

```
-----
ValueError                                Traceback (most recent call
/home/cesitar/<ipython-input-8-20d5230b6aed> in <module>()
----> 1 numpy.concatenate((a,c))
ValueError: arrays must have same number of dimensions
>>> numpy.concatenate((b,c))
array([[ 1.,  2.,  3.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Otras alternativas para cambiar la "forma" de un array

Función

Descripción

<code>numpy.hstack(seq)</code>	Apila los arrays de la secuencia "seq" horizontalmente (agrega columnas).
<code>numpy.vstack(seq)</code>	Apila los arrays de la secuencia "seq" verticalmente (agrega filas).

```
>>> a = numpy.array([[1, 2, 3], [4, 5, 6]])
>>> b = numpy.zeros(2)
>>> c = numpy.arange(8).reshape((2,4))
>>> print numpy.vstack((a.T, b))
[[ 1.  4.]
 [ 2.  5.]
 [ 3.  6.]
 [ 0.  0.]]
>>> print numpy.hstack((a, c))
[[1 2 3 0 1 2 3]
 [4 5 6 4 5 6 7]]
```


Otras alternativas para cambiar la "forma" de un array

Función

Descripción

`numpy.hsplit(array, escalar|seq)`

Divide array en subarrays por columnas. Misma forma de subarray si se da escalar. Corta por las columnas dadas en seq.

`numpy.vsplit(array, escalar|seq)`

Divide en subarrays por filas.

```
>>> a = numpy.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> print numpy.vsplit(a, 2)
...
ValueError: array split does not result in an equal division
>>> print numpy.vsplit(a, (1,2))
[array([[1, 2, 3]]), array([[4, 5, 6]]), array([[7, 8, 9]])]
>>> print numpy.hsplit(a, (1,))
[array([[1],
        [4],
        [7]])]
[array([[2, 3],
        [5, 6],
        [8, 9]])]
```

Operaciones aritméticas

Supongamos que quiero sumar un número a un array.
¿Cómo se hace?

Caso de listas PYTHON

```
>>> inicial = [1, 3, 6, -1]
>>> # sumar 3
>>> final = list()
>>> for elem in inicial:
...     final.append(elem + 3)
>>> final
[4, 6, 9, 2]
>>> o mas 'pythonicamente'
>>> final = [elem + 3 for elem in inicial]
```

Operaciones aritméticas

En el caso de arrays, la simplificación es notable.

```
>>> a_inicial = numpy.array([1, 3, 6, -1], dtype='int')
>>> a_inicial
array([ 1,  3,  6, -1])
>>> a_final = a_inicial + 3
>>> a_final
array([4, 6, 9, 2])
```

para listas PYTHON

```
>>> inicial = [1, 3, 6, -1]
>>> final = inicial + 3
TypeError: can only concatenate list (not "int") to list
```

Operaciones aritméticas

Podemos $+$, $-$, $*$ y $/$ cualquier array con un número.
Tendremos problemas si realizamos operaciones de valor no definido.

Resultados no definidos

```
>>> a = numpy.arange(4)
>>> print a
[0 1 2 3]
>>> b = a/0.
>>> print b
[ nan  inf  inf  inf]
```

Operaciones aritméticas

Entre arrays

Para arrays con **idénticas dimensiones**: Las operaciones se realizan entre cada par de elementos que ocupan la misma posición.

Resultados no definidos

```
>>> op1 = numpy.zeros([2,3]) + 3

>>> op2 = numpy.array([[0,1,2],[3,4,5]])

>>> op1
array([[ 3.,  3.,  3.],
       [ 3.,  3.,  3.]])

>>> op2
array([[0, 1, 2],
       [3, 4, 5]])

>>> op1 * op2
array([[ 0.,  3.,  6.],
       [ 9., 12., 15.]])
```

Operaciones aritméticas

Entre arrays

Cuando los arrays tienen formas y dimensiones diferentes hay que andarse con cuidado.

El término **broadcasting** se emplea para describir la forma en la que NUMPY trata las operaciones entre arrays de diferente shape.

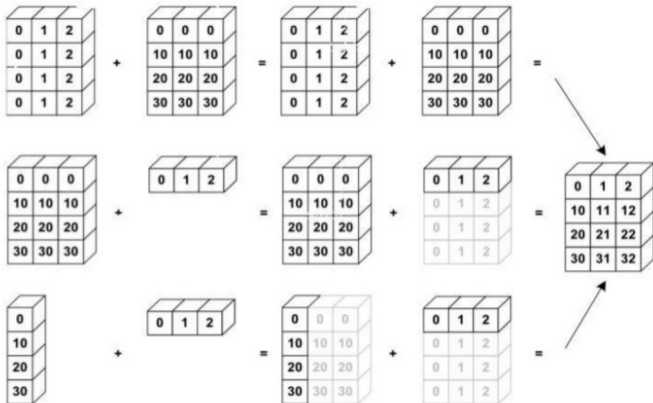
Reglas de broadcasting

- 1 El número de dimensiones no tiene por qué ser el mismo.
- 2 Es posible cuando las dimensiones finales son iguales.
- 3 Vale cuando una de las dimensiones finales es 1.

En caso de intentar hacer una operación imposible, se lanza una excepción del tipo **ValueError**.

Operaciones aritméticas

Entre arrays



Operaciones relacionales

- Son las que comparan un array con un dato simple o las que comparan arrays entre sí.
- El resultado es un array de valores booleanos (True/False).

Operadores relacionales

- $>$, $<$, $>=$, $<=$, (“mayor que”, “menor que”, “mayor o igual que”, “menor o igual que”)
- $==$ (“igual que”)
- $!=$ (“distinto que”)

Operaciones relacionales

Ejemplo 1

```
>>> a = numpy.arange(4)
>>> print a
[0 1 2 3]
>>> print a > 2
[False False False  True]
>>> print a >= 2
[False False  True  True]
>>> print a < 9
[ True  True  True  True]
>>> print a <= 1
[ True  True False False]
>>> b = numpy.array([0, 1, 3, 3])
>>> print a == a
[ True  True  True  True]
>>> print a == b
[ True  True False  True]
>>> print a != b
[False False  True False]
```

Operaciones Lógicas

- Son las que se dan entre datos (ya sean tipos simples o arrays) de tipo booleano.
- El resultado es un valor o array de tipo booleano (True/False).
- Se realizan **elemento a elemento**.

Operadores lógicos

- $\&$ (“y/AND lógico”)
- $|$ (“o/OR lógico”)
- \sim (“no/NOT lógico”). Este operador es **unario** (sólo necesita un operando).

Operaciones lógicas

Ejemplo 1

```
>>> a = numpy.arange(4)
>>> ab = a > 1
>>> print ab
[False False  True  True]
>>> b = numpy.ones(4, dtype=numpy.bool)
>>> print b
[ True  True  True  True]
>>> print ab & b
[False False  True  True]
>>> print ~ab
[ True  True False False]
>>> print ab | b
[ True  True  True  True]
```

Funciones relacionales y lógicas de NUMPY

Función	Descripción
Testeo de valor True	
<code>all(a[, axis])</code>	Comprueba si todos los elementos del eje "axis" del array son True.
<code>any(a[, axis])</code>	Comprueba si algún elemento del eje "axis" del array es True.
Relacionales	
<code>allclose(a, b[, rtol, atol])</code>	Devuelve True si los arrays tienen sus elementos iguales dentro de un intervalo de tolerancia.
<code>array_equal(a1, a2)</code>	True si dos arrays tienen la misma forma y elementos. False en caso contrario.
<code>greater(x1, x2)</code>	Devuelve el valor de la comparación ($x1 > x2$) elemento a elemento.
<code>greater_equal(x1, x2)</code>	Devuelve el valor de la comparación ($x1 \geq x2$) elemento a elemento.
<code>less(x1, x2)</code>	Devuelve el valor de la comparación ($x1 < x2$) elemento a elemento.
<code>less_equal(x1, x2)</code>	Devuelve el valor de la comparación ($x1 \leq x2$) elemento a elemento.
<code>equal(x1, x2)</code>	Devuelve ($x1 == x2$) elemento a elemento.
<code>not_equal(x1, x2)</code>	Devuelve ($x1 \neq x2$) elemento a elemento.

Funciones relacionales y lógicas de NUMPY

Función	Descripción
Lógicas	
<code>logical_and(x1, x2)</code>	Determina el valor de x1 AND x2 elemento a elemento.
<code>logical_or(x1, x2)</code>	Determina el valor de x1 OR x2 elemento a elemento.
<code>logical_not(x)</code>	Determina el valor del NOT x1 elemento a elemento.

Funciones NUMPY relacionales y lógicas

Ejemplo 1

```
>>> a = numpy.arange(4)
>>> b = numpy.array([0, 1, 2.2, 3.1])
>>> print numpy.allclose(a, b, atol=0.25)
True
>>> print numpy.allclose(a, b, atol=0.15)
False
>>> c = numpy.array([[False, False], [True, True]])
>>> print c
[[False False]
 [ True  True]]
>>> print numpy.all(c, axis=0)
[False False]
>>> print numpy.all(c, axis=1)
[False  True]
```

Funciones NUMPY relacionales y lógicas

Ejemplo 1

```
>>> a = numpy.arange(4)
>>> b = numpy.array([0, 1, 2.2, 3.1])
>>> print numpy.array_equal(a, b)
False
>>> print numpy.array_equal(a, a)
True
>>> print numpy.greater(a, b)
[False False False False]
>>> print numpy.greater_equal(a, b)
[ True  True False False]
>>> print numpy.less(a, b)
[False False  True  True]
>>> print numpy.less_equal(a, b)
[ True  True  True  True]
>>> print numpy.equal(a, b)
[ True  True False False]
>>> print numpy.not_equal(a, b)
[False False  True  True]
```

Consulta de tipos de datos

La forma más sencilla es a través de la propiedad `dtype`.

```
>>> a = numpy.arange(4)
>>> print a.dtype
int64
>>> a.dtype
dtype('int64')
>>> a.dtype.name
'int64'
```

Otras formas incluyen el uso de funciones NUMPY.

Consulta de tipos de datos

Funciones NUMPY

Función	Descripción
<code>iscomplex(x)</code>	Devuelve un array de booleanos, con True para los elementos complejos.
<code>iscomplexobj(x)</code>	Comprueba el tipo complejo para un array. True si es complejo.
<code>isreal(x)</code>	Devuelve un array de booleanos, con True en aquellas posiciones donde el elemento es real.
<code>isrealobj(x)</code>	Devuelve True si x no es de tipo complejo o un array de números complejos.
<code>isscalar(num)</code>	Devuelve True si el tipo de num es escalar.

```
>>> a = numpy.arange(4, dtype=numpy.complex)
>>> print numpy.iscomplex(a)
[False False False False]
>>> print numpy.iscomplexobj(a)
True
>>> print numpy.isreal(a)
[ True  True  True  True]
>>> print numpy.isrealobj(a)
False
>>> print numpy.isscalar(a)
False
>>> print numpy.isscalar(1)
```

Cambio de tipo de dato en un array

Procedimiento conocido como **casting** de tipos.

Imaginemos que leo un fichero y tengo el array

```
>>> a  
array(['1', '2', '3', '4'],  
      dtype='<S1')
```

¿Qué sucede si intento sumar el número entero 2?

```
>>> a + 2  
TypeError: unsupported operand  
type(s) for +: 'numpy.ndarray' and 'int'
```

Cambio de tipo de dato en un array

Lo que vemos es que tenemos la información que queremos, pero el tipo no es correcto.

Solucionamos cambiando el tipo. Procedimientos:

- 1 Volvemos a generar el array con
`numpy.array(..., dtype='typeNumpy')`
- 2 Usamos la función de NUMPY:
`numpy.typeNumpy(array)`
- 3 Usamos el método de los arrays:
`nombreakarray.astype('typeNumpy')`

`typeNumpy` es cualquiera de dados en la transparencia 17.

Cambio de tipo de dato en un array

Generamos nuevo array

```
>>> a.dtype
dtype('S1')
>>> a1 = numpy.array(a, dtype='int')
>>> a1.dtype
dtype('int64')
>>> # otra forma de escribir el tipo
>>> a2 = numpy.array(a, dtype=numpy.float32)
>>> a2.dtype
dtype('float32')
```

Cambio de tipo de dato en un array

Funciones/métodos de cambio de tipo

```
>>> a
array(['1', '2', '3', '4'],
      dtype='<S1')
>>> a3 = numpy.complex64(a)
>>> a3
array([ 1.+0.j,  2.+0.j,  3.+0.j,
        4.+0.j], dtype=complex64)
>>> a4 = a.astype('int8')
>>> a4.dtype
dtype('int8')
```

ESQUEMA

- 1 Métodos de entrada/salida
- 2 Ordenación
- 3 Búsquedas
- 4 Matemáticas
- 5 Estadística básica
- 6 Correlaciones y covarianzas
- 7 Muestreo aleatorio
- 8 Polinomios
- 9 Ajustes
- 10 Arrays con máscaras

Métodos de Entrada/Salida

```
ndarray.tolist()  
ndarray.tofile(fid[  
    ,sep,format])
```

Devuelve el array como una lista.

Escribe el array en un fichero, en formato texto o binario (defecto).

- fid, puede ser un objeto fichero con permiso de escritura o una cadena de caracteres con la ruta al fichero.

```
numpy.fromfile(file[  
    ,dtype='float',  
    count=-1, sep='')]
```

Lee ficheros escritos con *ndarray.tofile*

- fid, puede ser un objeto fichero con permiso de lectura o una cadena de caracteres con la ruta.

```
ndarray.dump(file)
```

- count, numero de lineas a leer (-1, todo el fichero)

- sep, separador de campos ("" indica fichero binario)

Vuelca la serialización del array al fichero "file". Se puede leer con el módulo pickles.

```
ndarray.dumps()
```

Devuelve la serialización del array como cadena de caracteres.

El formato es que usa el módulo pickle. `pickle.loads` o `numpy.loads` pueden convertir esa cadena en array.

Métodos de Entrada/Salida

El volcado de arrays en fichero es útil

- cuando se trabaja siempre con el mismo conjunto de datos (aceleramos su lectura)
- cuando se quiere almacenar datos durante la ejecución de un script.

Métodos de Entrada/Salida

Ejemplo de I/O

```
>>> a = numpy.arange(0, 1, 0.2)
>>> a.tolist()
[0.0, 0.2, 0.4, 0.60000000000000001, 0.8]
>>> a.tofile('datos.dat', ';', "%4.2f")
# contenido fichero: "0.00;0.20;0.40;0.60;0.80"
>>> a.tofile('binario.dat')
# contenido fichero: "\00\00\00\00\00\00\00\00\00\9A\99\99\99\99\99\C9?"
>>> b = numpy.fromfile('binario.dat')
>>> b
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

Métodos de Entrada/Salida

Ejemplo de I/O

```
>>> a = numpy.arange(0, 1, 0.2)
>>> a.dump('datos.pick') # volcado de datos a fichero
>>> import pickle
>>> f = open('datos.pick', 'r')
>>> b = pickle.load(f)
>>> f.close()
>>> b
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
>>> a.dumps()
'\x80\x02numpy.core.multiarray\n_reconstruct\nq\x01numpy\nndarray\n'
>>> pickle.loads(a.dumps())
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

Métodos de Entrada/salida

Función `numpy.genfromtxt`

Genérica y potente de carga de datos desde fichero en formato texto.

```
numpy.genfromtxt(fname, dtype='float', comments='#',  
    delimiter=None, skip_header=0, skip_footer=0,  
    converters=None, missing='', missing_values=None,  
    filling_values=None, usecols=None, names=None,  
    excludelist=None, deletechars=None,  
    replace_space='_', autostrip=False,  
    case_sensitive=True, defaultfmt='f%i', unpack=None,  
    usemask=False, loose=True, invalid_raise=True)
```

Esto es lo más en cuanto a lectura de ficheros en un formato no nativo PYTHON/NUMPY.

Métodos de Entrada/salida

Función `numpy.genfromtxt`

Parámetros más importantes:

- `fname`, nombre del fichero.
- `dtype`, tipo de datos a leer. Si se da el valor `None`, la función elige el tipo que mejor se ajuste a cada columna.
- `comments`, carácter usado como comentario. Todo lo que venga detrás se ignora.
- `delimiter`, cadena usada para separar campos.
- `skip_header`, número de líneas a ignorar al principio del fichero.
- `skip_footer`, número de líneas a ignorar al final del fichero.
- `missing_values`, conjunto de cadenas que corresponden a valores perdidos.
- `filling_values`, conjunto de valores usados para sustituir a los valores perdidos.
- `usecols`, (lista/tupla) índices de las columnas que se van a usar (empiezan en 0).
- `names`, (`None`—`True`—`string`—`sequence`) Nombres con los que identificar las columnas. Si es `True`, lee la primera columna tras `skip_header` y toma los nombre de allí. si es una cadena, los nombres se separan por comas.

Métodos de Entrada/salida

Función `numpy.genfromtxt`

Parámetros más importantes:

- `excludelist`, lista de nombres a excluir. Ya incluye `['return','file','print']`.
- `deletechars`, cadena con caracteres a eliminar de los nombres.
- `autostrip`, booleano que indica si se quitan espacios al principio y al final de las cadenas de texto.

Función `numpy.genfromtxt`

Ejemplo 1

- Para los ejercicios que siguen, usaremos los datos del fichero *NucleosPoblacion.csv*, disponible en la web del curso.
- Es un fichero en formato texto.
- la primera línea contiene los nombres de los campos.
- El carácter separador es ';'.

```
>>> data =numpy.genfromtxt('NucleosPoblacion.csv', delimiter=';', \
dtype=None, names=True)
>>> data.dtype.names # acceso a nombres de columnas
('FID', 'OBJECTID', 'CodINE', 'Texto', 'Poblacion', 'CodMun',
'Municipio', 'CodProvin', 'Provincia', 'X', 'Y')
>>> data['Poblacion'].dtype # acceso a cada array por columnas
dtype('float64')
>>> data['Municipio'].size
852
```

Ordenación de arrays

- Toda la información se maneja mejor cuando hay algún tipo de ordenación.
- NUMPY proporciona rutinas para ordenar arrays, sean del tipo o dimensión que sean.

Función	Descripción
<code>numpy.sort(a, axis=-1)</code>	Devuelve un array el mismo tipo y forma, ordenado de menor a mayor según el eje "axis". Si es None, transforma el array a 1D y ordena.
<code>numpy.argsort(a, axis=-1)</code>	Devuelve un <u>array de posiciones</u> que ordenaría el original de menor a mayor según "axis" dado. Si axis=None, transforma a 1D y retorna los índices.

NOTA:

- En un array 2D, axis=0 corresponde a filas, axis=1 a columnas.
- En un array 3D, axis=0 corresponde a profundidad, axis=1 a filas y axis=2 a columnas.

Ordenación de arrays

Ejemplo 1

```
>>> a = numpy.array([3, 2, 4, 1, -1], dtype='int')
>>> b = numpy.sort(a)  # no cambia el array
>>> b
array([-1,  1,  2,  3,  4])
>>> index = numpy.argsort(a)
>>> index
array([4, 3, 1, 0, 2])
>>> a[index]  # Atención a la seleccion
array([-1,  1,  2,  3,  4])
```


Búsqueda en arrays

Se buscan elementos en función de uno o varios criterios. Esos criterios se combinan mediante funciones u operadores lógicos, resultando en un array de booleanos.

```
numpy.where(condición,  
            a1, a2)
```

- `condición` es un array de booleanos.
Devuelve un array con los elementos de `a1` donde la `condición` es cierta y los de `a2` si es falsa.
Los 3 parámetros son arrays de la misma forma (`shape`), o compatibles según reglas de [broadcast](#).

```
numpy.extract(condición, a)
```

Devuelve los elementos de array donde `condición` es `True`. Devuelve siempre un array **unidimensional**, independientemente de la dimensión del array inicial. “`condición`” y “`a`”, deben tener la misma forma.

```
numpy.compress(condición, a,  
               axis=None)
```

Es la generalización de `extract`.
Retorna un array n-dimensional. La `condición` puede aplicarse a una u otras dimensiones, según el array “`a`” de entrada.

Si la longitud de la `condición` es menor que el tamaño del array en ese eje, la salida se trunca a la longitud de la `condición`.

Búsqueda en arrays

Ejemplo 1

```
>>> arr = numpy.arange(12).reshape((3, 4))
>>> arr
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> condition = numpy.mod(arr, 3)==0 # multiplo de 3
>>> condition
array([[ True, False, False,  True],
       [False, False,  True, False],
       [False,  True, False, False]], dtype=bool)
>>> numpy.extract(condition, arr)
array([0, 3, 6, 9])
```

Búsqueda en arrays

Ejemplo 2

```
>>> a = numpy.array([[1, 2], [3, 4], [5, 6]])
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> numpy.compress([0, 1], a, axis=0)
array([[3, 4]])
>>> numpy.compress([False, True, True], a, axis=0)
array([[3, 4],
       [5, 6]])
>>> numpy.compress([False, True], a, axis=1)
array([[2],
       [4],
       [6]])
```

Búsqueda en arrays

Valores especiales

Función	Descripción
<code>argmax(a[, axis=None])</code>	Devuelve los índices de los valores iguales al máximo según el eje "axis". Si <code>axis=None</code> buscan en todo el array.
<code>nanargmax(a[, axis=None])</code>	Devuelve los índices de los valores iguales al máximo en el eje dado por "axis". Ignora los NaNs. (<code>axis=None</code> , todo el array.)
<code>argmin(a[, axis=None])</code>	Devuelve los índices de los valores iguales al mínimo según el eje "axis". (<code>axis=None</code> , en todo el array.)
<code>nanargmin(a[, axis=None])</code>	Devuelve los índices de los valores iguales al mínimo según el eje "axis". Ignora los NaNs. (<code>axis=None</code> , en todo el array.)
<code>nonzero(a)</code>	Devuelve una tupla de arrays, una por cada dimensión de <code>a</code> , conteniendo los índices de los elementos que no son cero en esa dimensión.
<code>flatnonzero(a)</code>	Devuelve los índices de los elementos que no son cero en la versión 1D del array <code>a</code> .

Búsqueda en arrays

Función	Descripción
<code>nonzero(a)</code>	Devuelve los índices de los elementos que no son cero.
<code>isnan(a)</code>	Devuelve un array de booleanos: <ul style="list-style-type: none">- <i>True</i> en las posiciones donde hay nan- <i>False</i> en caso contrario.
<code>isinf(a)</code>	Devuelve un array de booleanos: <ul style="list-style-type: none">- <i>True</i> en las posiciones donde hay valores infinite- <i>False</i> en caso contrario.

Valores especiales

Ejemplo 1

```
>>> b = numpy.arange(10)
>>> b = b.reshape((2,5))
>>> b
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> numpy.argmax(b)
9
>>> numpy.argmax(b, axis=0)
array([1, 1, 1, 1, 1])
>>> numpy.argmax(b, axis=1)
array([4, 4])
>>> d = numpy.arange(4)
>>> e = d / 0.
>>> e
array([ nan,  inf,  inf,  inf])
>>> numpy.isnan(e)
array([ True, False, False, False], dtype=bool)
>>> numpy.isinf(e)
array([False,  True,  True,  True], dtype=bool)
```

Operaciones matemáticas

Trigonómicas

`sin(x)`
`cos(x)`
`tan(x)`
`arcsin(x)`
`arccos(x)`
`arctan(x)`
`hypot(x, y)`
`degrees(x)`
`radians(x)`
`deg2rad(x)`
`rad2deg(x)`

Redondeo

`around(a[, decimals])`
`round_(a[, decimals,])`
`rint(x)`
`fix(x)`
`floor(x)`
`ceil(x)`
`trunc(x)`

Hyperbólicas

`sinh(x)`
`cosh(x)`
`tanh(x)`
`acrsinh(x)`
`arctan(x)`

Exp & Logs

`exp(x)`
`exp1m(x)`
`exp2(x)`
`log(x)`
`log10(x)`
`log2(x)`
`log1p(x)`

Miscelánea

`sqrt(x)`
`power(x)`
`fbas(x)`
`sign(x)`
`nan_to_num(x)`

Sum. & Prod. & Diff.

`prod(a[, axis])`
`sum(a[, axis])`
`nansum(a[, axis])`
`cumprod(a[, axis])`
`cumsum(a[, axis])`
`gradient(f, *varargs)`
`cross(a, b)`

x e y son arrays. Las operaciones se realizan elemento a elemento. Las entradas a funciones trigonométricas que requieran de ángulos se dan en radianes.

REFERENCIA: <http://docs.scipy.org/doc/numpy/reference/routines.math.html>

Estadística básica

De ordenación

Función	Descripción
<code>numpy.amin(a, axis=None)</code>	Devuelve un array (o escalar) con el valor mínimo del array a lo largo del eje dado por "axis".
<code>numpy.amax(a, axis=None)</code>	Devuelve un array (o escalar) con el valor máximo del array a lo largo del eje dado por "axis".
<code>numpy.nanmin(a, axis=None)</code>	Devuelve un array (o escalar) con el valor mínimo del array a lo largo del eje dado por "axis".
<code>numpy.nanmax(a, axis=None)</code>	Ignora los valores NaN. Devuelve un array (o escalar) con el valor máximo del array a lo largo del eje dado por "axis".
<code>numpy.ptp(a, axis=None)</code>	Ignora los valores NaN. Devuelve el rango de valores (máximo - mínimo) en el "axis" dado. El nombre de esta función viene del acrónimo "peak to peak".
<code>numpy.percentile(a, q, axis=None)</code>	Calcula y devuelve el percentil q-ésimo del array a en el eje "axis" especificado. q (escalar) en $[0,100]$.

NOTA: Si `axis=None`, se trabaja sobre la versión transformada a 1D del array, devolviendo un único valor (escalar) para todo el array.

Estadística básica

Ejemplo 1

```
>>> a = numpy.array([[5, 6, 1], [2, 3, 8]])
>>> print a
[[5 6 1]
 [2 3 8]]
>>> print numpy.amax(a)
8
>>> print numpy.amax(a, axis=0)
[5 6 8]
>>> print numpy.amax(a, axis=1)
[6 8]
>>> print numpy.percentile(a, 25)
2.25
>>> print numpy.percentile(a, 25, axis=0)
[ 2.75  3.75  2.75]
>>> print numpy.percentile(a, 25, axis=1)
[ 3.   2.5]
>>> print numpy.ptp(a)
7
>>> print numpy.ptp(a, axis=1)
[5 6]
```

Estadística básica

Ejemplo 2

```
>>> b = numpy.array([[5, numpy.nan, 1], [2, 3, numpy.nan]])
>>> print numpy.amin(b)
nan
>>> print numpy.amax(b)
nan
>>> print numpy.nanmin(b)
1.0
>>> print numpy.nanmin(b, axis=0)
[ 2.  3.  1.]
>>> print numpy.nanmin(b, axis=1)
[ 1.  2.]
>>> print numpy.nanmax(b, axis=1)
[ 5.  3.]
>>> print numpy.ptp(b)
nan
>>> print numpy.ptp(b, axis=0)
[ 3.  nan  nan]
>>> print numpy.ptp(b, axis=1)
[ nan  nan]
```

Estadística básica

¿Cómo obviar el problema de los NaN?

Posibles soluciones:

- Seleccionar el sub-array de elementos que no son NaN.
- Usar arrays con máscara (*masked arrays*) (más adelante).

Estadística básica

Valores promedio y varianzas

Función	Descripción
<code>numpy.average(a, axis=None, weights=None)</code>	Devuelve un escalar o array con la media “pesada” del array <i>a</i> por los valores “weights” en el eje “axis” seleccionado. Los pesos pueden ser arrays 1-D, en cuyo caso ha de tener la misma longitud que <i>a</i> en el eje seleccionado. Si <code>weights=None</code> se asume el mismo peso (valor=1) para todos los elementos.
<code>numpy.mean(a, axis=None, dtype=None)</code>	Devuelve un escalar o array con la media aritmética del array sobre el “axis” dado. “dtype” establece el tipo de datos de entrada sobre el que promediar. El valor asignado por defecto es el del tipo del array.
<code>numpy.median(a, axis=None)</code>	Devuelve un escalar o array con la mediana del array para el eje seleccionado.
<code>numpy.std(a, axis=None, dtype=None, ..., ddof=0)</code>	Devuelve un escalar o array con la desviación estándar en el eje seleccionado. <code>ddof</code> es el acrónimo de <i>Delta Degrees of Freedom</i> . El denominador usado en los cálculos es $N - ddof$, donde N es el número de elementos.
<code>numpy.var(a, axis=None, dtype=None, ..., ddof=0)</code>	Devuelve un escalar o array con la varianza de los elementos del array en el eje seleccionado. Misma leyenda que <code>std</code> para el resto de parámetros.

Estadística básica

Ejemplo 3

```
>>> a = numpy.arange(4).reshape((2, 2))
>>> a
array([[0, 1],
       [2, 3]])
>>> print numpy.average(a, axis=1, weights=(0, 1))
[ 1.  3.]
>>> print numpy.mean(a, axis=0)
[ 1.  2.]
>>> print numpy.mean(a, axis=1)
[ 0.5  2.5]
>>> print numpy.median(a)
1.5
>>> print numpy.median(a, axis=1)
[ 0.5  2.5]
>>> print numpy.std(a)
1.11803398875
>>> print numpy.var(a)
1.25
>>> print numpy.std(a) * numpy.std(a)
1.25
```

Estadística básica

Histogramas

Función

```
numpy.histogram(a, bins=10,
               range=None, normed=False,
               weights=None, density=None)
```

```
numpy.histogram2d(x, y,
                 bins=10, range=None,
                 normed=False, weights=None)
```

Descripción

Devuelve una tupla con dos arrays: (histograma, bins) de la versión 1D del array. bins, puede ser un número, lista o array (con intervalos no uniformes).

Si normed=True, retorna el histograma normalizado.

Pueden “pesarse” los elementos del array.

Si weights=None todos los elementos cuentan como 1.

Si no, debe ser un array de la misma forma que a.

normed afecta también a weights.

density es un booleano. Si es False el histograma devuelve el número de elementos por cada *bin*. Si no devuelve el valor de la función densidad de probabilidad de cada bin, normalizada a todo el rango de *bins*.

Devuelve la tupla de arrays: (histo, bins_x, bins_y) x e y son arrays. bins puede tomar:

- un valor *entero* (núm. divisiones en cada dimensión),
- [*entero, entero*] (num. divisiones en cada eje),
- un *array* (válido para los dos ejes) o
- [*array, array*], uno para cada dimensión.

Estadística básica

Histogramas

Función

`numpy.digitize(a, bins,
right=False)`

Descripción

Devuelve un array de índices. Cada índice indica a qué intervalo de “bins” pertenece el elemento del array.

“bins” debe ser monótono creciente o decreciente.

Si los valores de *a* están fuera del rango de “bins”, devuelve 0 para ese elemento.

Si *right=True* el intervalo de “bins” es cerrado.

`numpy.bincount(x,
weights=None, minlength=None)`

Devuelve el número de ocurrencias de cada valor del array en cada uno de los intervalos “bin” (por defecto de anchura igual a 1).

Estadística básica

Histogramas: Ejemplo 1

Ejemplos de uso de histogramas

```
>>> a = numpy.array([0, 1, 2, 3, 2, 4, 3, 3, 3, 5, 6, 9])
>>> bins = numpy.arange(10)
>>> numpy.histogram(a, bins)
(array([1, 1, 2, 4, 1, 1, 1, 0, 1]),
 array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]))
>>> numpy.digitize(a, bins)
array([ 1,  2,  3,  4,  3,  5,  4,  4,  4,  6,  7, 10])
```


Correlaciones y covarianzas

Covarianza

“En probabilidad y estadística, la covarianza es un valor que indica el grado de variación conjunta de dos variables aleatorias. Es el dato básico para determinar si existe una dependencia entre ambas variables y además es el dato necesario para estimar otros parámetros básicos, como el coeficiente de correlación lineal o la recta de regresión.”

Correlación

“En probabilidad y estadística, la correlación indica la fuerza y la dirección de una relación lineal y proporcionalidad entre dos variables estadísticas. Se considera que dos variables cuantitativas están correlacionadas cuando los valores de una de ellas varían sistemáticamente con respecto a los valores homónimos de la otra: si tenemos dos variables (A y B) existe correlación si al aumentar los valores de A lo hacen también los de B y viceversa. La correlación entre dos variables no implica, por sí misma, ninguna relación de causalidad.”

FUENTE: Wikipedia.

Correlaciones y covarianzas

Función	Descripción
<code>numpy.cov(m, y=None, rowvar=1, bias=0, ddof=None)</code>	<p>Estima la matriz de covarianza. <i>m</i> es un array 1D o 2D. Cada columna de <i>m</i> representa una observación. Cada fila, una variable. <i>y</i> es un conjunto adicional de observaciones (misma forma que <i>m</i>). Si <i>rowvar</i> es distinta de cero, cada fila representa una variable. Cada columna una observación. <i>bias</i> es un entero. Si es cero, se trasponen filas y columnas. La normalización por defecto es $(N - 1)$, donde N es el núm. de observaciones. Si <i>bias</i> = 1 la normalización es N. <i>ddof</i> sobrescribe a <i>bias</i>. Si es distinto de <i>None</i>, la normalización es $(N - ddof)$.</p>
<code>numpy.correlate(a, v, mode='valid')</code>	<p>Correlaciona dos arrays 1D según la expresión $z[k] = \sum_n a[n] * conj(v[n + k])$ <i>mode</i> vale 'full', 'valid' o 'same', que establece los límites del desplazamiento (<i>n</i>) entre arrays (ver documentación).</p>
<code>numpy.corrcoef(x, y=None, rowvar=1, bias=0, ddof=None)</code>	<p>Devuelve los coeficientes de correlación $P_{ij} = \frac{c_{ij}}{\sqrt{c_{ii} \cdot c_{jj}}}$. <i>x</i> es un array 1D o 2D. Cada fila es una variable. Cada columna una observación. Para el resto, misma leyenda que <code>numpy.cov</code>.</p>

Estadística básica

Correlaciones y covarianzas: Ejemplo 1

```
>>> x = [-2.1, -1, 4.3]
>>> y = [3, 1.1, 0.12]
>>> X = numpy.vstack((x,y))
>>> print X
[[-2.1 -1.    4.3 ]
 [ 3.    1.1  0.12]]
>>> print numpy.cov(X)
[[ 11.71         -4.286         ]
 [ -4.286         2.14413333]]
>>> print numpy.cov(x, y)
[[ 11.71         -4.286         ]
 [ -4.286         2.14413333]]
>>> print numpy.cov(y, x)
[[ 2.14413333  -4.286         ]
 [ -4.286         11.71         ]]
>>> print numpy.cov(x)
11.71
>>> print numpy.var(x, ddof=1)
11.71
```

Estadística básica

Correlaciones y covarianzas: Ejemplo 2

```
>>> x = [-2.1, -1, 4.3]
>>> y = [3, 1.1, 0.12]
>>> X = numpy.vstack((x, y)) # agrega filas
>>> print X
[[-2.1  -1.    4.3 ]
 [ 3.    1.1   0.12]]
>>> print numpy.cov(X)
[[ 11.71      -4.286      ]
 [ -4.286      2.14413333]]
>>> print numpy.correlate(x, y)
[-6.884]
>>> print numpy.corrcoef(x, y)
[[ 1.          -0.85535781]
 [-0.85535781  1.          ]]
```

Una variable correlaciona perfectamente consigo misma (valor 1 en la diagonal principal). La correlación varía en $[-1, 1]$.

Muestreo aleatorio

Hemos de importar de forma explícita este módulo.

```
import numpy.random
```

Función	Descripción
<code>rand(d0, d1, ..., dn)</code>	Devuelve una array de dimensión (d0, d1, ..., dn) con números aleatorios en [0, 1).
<code>randn(d0, d1, ..., dn)</code>	Muestra de dimensión (d0, d1, ..., dn) elementos de una distribución normal estándar.
<code>randint(low, high=None, size=None)</code>	Devuelve "size" num enteros en [start, end). size puede ser entero o secuencia (ND-array).
<code>random_integers(low, high=None, size=None)</code>	Igual que la anterior pero en [start, end].
<code>choice(a, size=1, replace=True, p=None)</code>	Genera una muestra aleatoria de tamaño size con los elementos del array 1D a. Si replace=True la muestra es con repetición. p es un array (de la misma dimensión que a) que indica las probabilidades de cada elemento de a.
<code>shuffle(x)</code>	Mezcla la secuencia dada por x. Cambia x. Devuelve None.
<code>permutation(x)</code>	Permuta de forma aleatoria la secuencia x. Si x es n-dimensional, sólo permuta la dimensión dada por el primer índice.

Muestreo aleatorio

Ejemplo 1

```
>>> import numpy.random
>>> print numpy.random.rand(2, 3)
[[ 0.47817202  0.44355165  0.9863675 ]
 [ 0.72560495  0.11331353  0.21659595]]
>>> print numpy.random.randn(4)
[-0.98550123  0.74578984 -2.04972696  0.36567011]
>>> print numpy.random.randint(0, 11, size=(2, 3))
[[10  5  0]
 [ 7  1  7]]
>>> print numpy.random.random_integers(0, 3, 6)
[1 3 1 0 1 1]
>>> a = numpy.arange(5)
>>> print numpy.random.choice(a, size=4)
[3 1 2 1]
>>> print numpy.random.choice(a, size=4, replace=False)
[2 0 3 4]
```

Muestreo aleatorio

Ejemplo 2

```
>>> a = numpy.arange(5)
>>> numpy.random.shuffle(a)
>>> a
[3 2 0 4 1]
>>> b = numpy.arange(9).reshape((3, 3))
>>> print numpy.random.permutation(b)
[[6 7 8]
 [3 4 5]
 [0 1 2]]
```

Polinomios

Se gestionan a través de la clase “Polynomial” del paquete `numpy.polynomial.polynomial` que debe ser importado.

Creación

```
numpy.polynomial.polynomial.Polynomial(coef)
```

que devuelve un objeto de tipo “Polynomial”.

“coef” es una tupla que establece los coeficientes del polinomio, en orden creciente. Esto es, (1, 2, 3) representa al polinomio $1 + 2 * x + 3 * x^2$.

```
>>> import numpy.polynomial.polynomial as npp
>>> pol = npp.Polynomial([1, 2, 3])
>>> print pol
poly([ 1.  2.  3.]
```


Polinomios

Creación alternativa

Se puede generar un polinomio si se conocen sus raíces. A partir de una función situada en el módulo `numpy.polynomial.polynomial`.

Función	Descripción
<code>polyfromroots(roots)</code>	Genera un polinomio a partir de sus raíces, mediante el producto $(x - r[0]) * (x - r[1]) * \dots * (x - r[n - 1])$ Atención al signo negativo en el producto.

```
>>> import numpy.polynomial.polynomial as npp
>>> pol2 = npp.Polynomial.fromroots([1, -2])
>>> print pol2
poly([-2.  1.  1.])
```

Polinomios

Utilidad

¿Qué se puede hacer con polinomios?

- 1 Buscar sus raíces.
- 2 Derivarlos e integrarlos.
- 3 Operar entre ellos.
- 4 Realizar ajustes a datos obtenidos en una muestra.
- 5 Evaluación de su valor en un punto.

Polinomios

Métodos

Método	Descripción
<code>copy()</code>	Devuelve una copia del polinomio.
<code>degree()</code>	Devuelve el grado del polinomio.
<code>cutdeg(deg)</code>	Devuelve un polinomio con grado reducido a “deg”, descartando términos de orden superior.
<code>roots()</code>	Devuelve la lista de raíces del polinomio.
<code>deriv(m=1)</code>	Calcula la derivada <i>m-ésima</i> del polinomio.
<code>integ(m=1, k=[], lbound=None)</code>	Devuelve la integral del polinomio. <i>m</i> es el número de integraciones. <i>k</i> es la lista de constantes de integración (de menor o igual tamaño que <i>m</i>). <i>lbound</i> es el límite inferior de la integral (si es definida).

NOTA: Un método es una función que se llama desde un objeto de una clase. La sintaxis sería: `objeto.metodo([parametros])`.

Es decir, si creo un objeto polinomio llamado `pol`, para obtener el grado del polinomio ejecutaré

```
pol.degree()
```

Operaciones entre polinomios

Siendo $c1$ y $c2$ las secuencias que definen los coeficientes de los polinomios, se pueden realizar las siguientes operaciones:

Función	Descripción
<code>polyadd(c1, c2)</code>	Devuelve la suma dos polinomios.
<code>polysub(c1, c2)</code>	Devuelve el polinomio resta de $c1$ y $c2$.
<code>polymul(c1, c2)</code>	Devuelve el producto de dos polinomios.
<code>polymulx(c)</code>	Devuelve el producto de un polinomio por x .
<code>polydiv(c1, c2)</code>	Devuelve una tupla con dos arrays: el que define el cociente y el que define el resto del resultado de dividir el polinomio $c1$ entre $c2$.
<code>polypow(c, pow)</code>	Devuelve el resultado de multiplicar el polinomio c al entero pow .

Operaciones entre polinomios

Ejemplo 1

```
>>> import numpy.polynomial.polynomial as npp
>>> c1 = [0, 2, 3]
>>> c2 = [0, 1]
>>> print npp.polyadd(c1, c2)
[ 0.  3.  3.]
>>> print npp.polysub(c1, c2)
[ 0.  1.  3.]
>>> print npp.polymul(c1, c2)
[ 0.  0.  2.  3.]
>>> print npp.polydiv(c1, c2)
(array([ 2.,  3.]), array([ 0.]))
```

Polinomios

Ajuste

Se puede generar un polinomio mediante un ajuste por mínimos cuadrados a una nube de puntos en 2D. Se usa el módulo `numpy.polynomial.polynomial`.

Función	Descripción
<code>polyfit(x, y, deg,)</code> <code>..., full=False, w=None)</code>	Retorna un polinomio de grado “deg”. x e y tienen que tener la misma dimensión final. Si y es 2D, se realizan múltiples ajustes (uno por cada columna de y). El resultado es también 2D (cada columna se refiere a un ajuste). “deg” es el grado del polinomio a ajustar. “full” se refiere al formato de salida: False, sólo coeficientes, True, coeficientes e información de diagnóstico ([residuals, rank, singular_values, rcond]). “w”, se refiere al peso de cada par de coordenadas (x_i, y_i). Si no se da, los pesos valen 1.

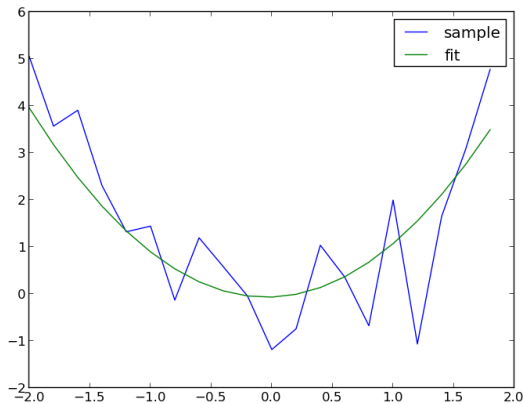
Ajuste

Ejemplo 1

```
>>> x = numpy.arange(-2, 2, 0.2)
>>> y = numpy.power(x, 2)
>>> noise = numpy.random.randn(x.size)
>>> res = npp.polyfit(x, y + noise, 2, full=True, \
w=1/numpy.power(noise, 2))
>>> print res
(array([-0.06632142,  0.08804425,
 1.0504697 ]), [array([ 78.97488008]), 3, array([ 1.73203349,
 0.00721654,  0.00281464]), 4.4408920985006262e-15])
```

Creación de array por ajuste

Ejemplo 1



Polinomios

Evaluación de valor en un punto

Dado un polinomio en la forma

$$p(x) = c_0 + c_1 \cdot x + c_2 \cdot x^2 + \cdots + c_n \cdot x^n$$

podemos evaluar un polinomio para cualquier valor de la variable independiente.

Función	Descripción
<code>polyval(x, c)</code>	Evalúa el polinomio dado por la secuencia <code>c</code> en los puntos dados por <code>x</code> (escalar o secuencia). <code>c</code> puede ser <i>1D</i> o <i>nD</i> . En este último caso hay que considerar más parámetros de la función <code>polyval</code> (ver documentación).

Evaluación

Ejemplo 1

Como ejemplo, evaluaremos el polinomio

$$p(x) = 2 + x^2$$

```
>>> import numpy.polynomial.polynomial as npp
>>> # Atención al orden de los coeficientes del polinomio
>>> npp.polyval(0, [2, 0, 1])
1.0
>>> npp.polyval([-1, 0, 1, 2], [1, 0, 1])
array([ 2.,  1.,  2.,  5.])
```

Polinomios especiales

NUMPY proporciona soporte a través del módulo `numpy.polynomial.polynomial` para la generación y operación con polinomios especiales:

- 1 Chebyshev
- 2 Legendre
- 3 Laguerre
- 4 Hermite

Para más información, consulte la documentación.

REFERENCIA: <http://docs.scipy.org/doc/numpy/reference/routines.polynomials.package.html>