Módulo 1 Numpy

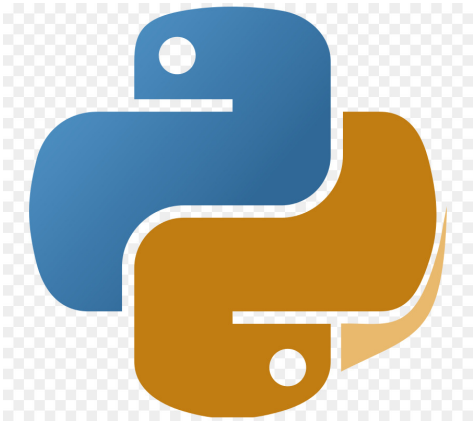
Curso de Python 3 Científico

César Husillos Rodríguez

IAA-CSIC

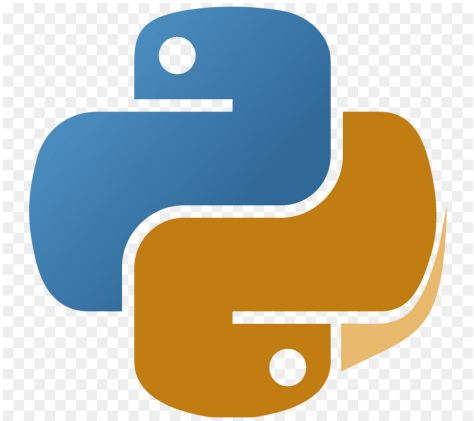
30 sep - 04 oct de 2019

Índice



Índice extendido

Índice



Consideraciones previas

Versión de NUMPY

- La versión última versión estable de NUMPY es la `v1.17.1`.
- El código del curso se ha probado con la versión `v1.16.4`.

Si tiene algún problema relacionado con la compatibilidad, puede actualizar su versión usando el comando `pip`.

`sudo pip install --upgrade numpy` (en LINUX)

Si no tiene `pip` instalado pruebe a ejecutar los comandos

```
sudo apt-get install python-pip python-dev
                     build-essential
sudo pip install --upgrade pip (en LINUX también)
```

Consideraciones previas

Lo obvio

Estas transparencias son sólo un resumen de `NUMPY`.

- Muchas de las funciones que se mencionan aquí, disponen de más parámetros que los mostrados. Hemos seleccionado los fundamentales.
- Seguramente, surgirán múltiples dudas a lo largo de la explicación. Como programadores con experiencia que somos en PYTHON, es conveniente que manejemos con soltura la ayuda, bien desde el intérprete o a través de la web.

Consideraciones previas

Uso de la ayuda

Recordatorio sobre la consulta de ayuda

- Una forma de acceder a la documentación desde el intérprete de PYTHON es con el comando `help`:
  `help(modulo)` o `help(modulo.función)`
- Las webs de referencia son:
  `http://www.python.org/doc/` (PYTHON)
  `http://docs.scipy.org/doc/numpy/reference/` (NUMPY)

Editando código

Línea de comandos

- Para ejecutar de forma interactiva usamos el intérprete de python (comando `python`).
- Si tenemos la posibilidad de instalar `ipython` tendremos mucho ganado: por ejemplo, la opción de **autocompletado** elimina muchos de los errores que se producen al escribir código.

Editando código

Editores de texto

- Cuando el script tiene que hacer operaciones complejas, está compuesto por muchas líneas, o se quiere ejecutar múltiples veces con diferentes parámetros de entrada, lo más cómodo es escribir el programa en un fichero de texto con la extensión *.py*.
- Otra ventaja que supone esta forma de trabajo es la corrección de errores, cambiando sólo la línea donde se da el error sin tener que reescribir todo el código.
- Los editores disponibles dependen de la plataforma: `emacs`, `gedit`, `notepad`... Cada programador tiene sus preferencias.

Editando código

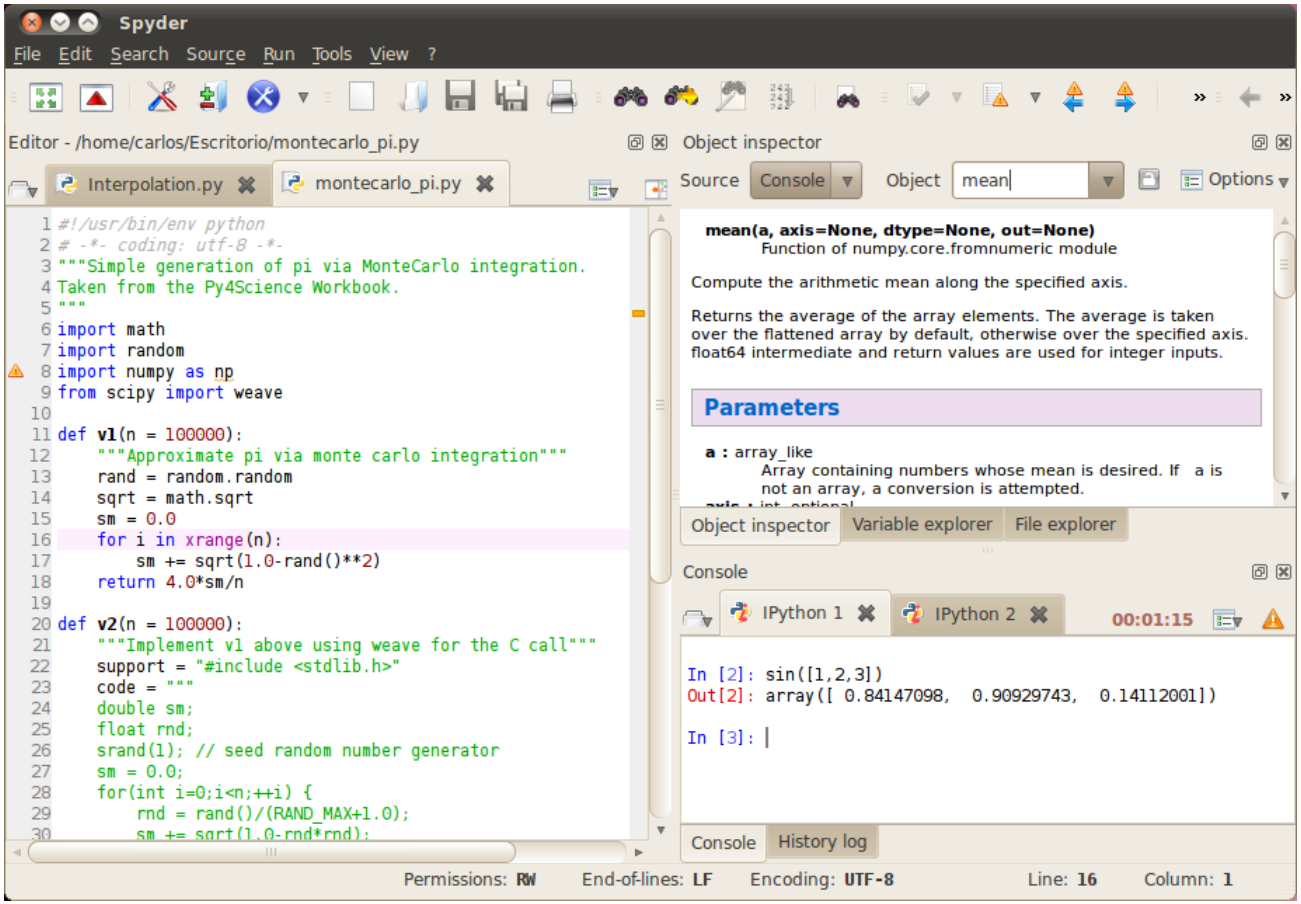
Entornos de Desarrollo Integrado (IDE)

- Son aplicaciones que facilitan la generación, depurado y ejecución de código.
- Disponen de: editor con resaltado de sintaxis, autocompletado de comandos, ayuda dinámica, capacidad de ejecutar el script o unas pocas líneas, acceso a consola interactiva de PYTHON, acceso a variables de programa y de entorno...
- Un ejemplo de este tipo de programas es `spyder`. Recomiendo su uso. Es descargable en múltiples plataformas desde el enlace

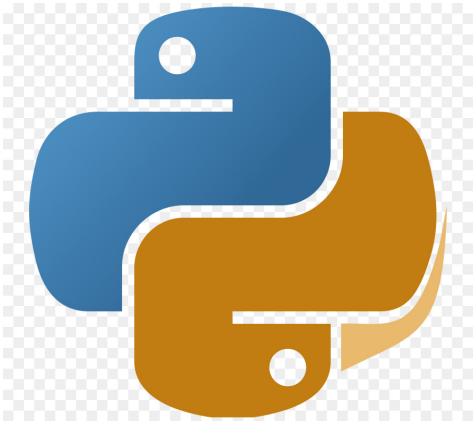
`https://www.spyder-ide.org`

Editando código

Entornos de desarrollo integrado (IDE)



Índice



Introducción

¿Qué es NUMPY?

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

1. Tipos de datos
2. Clases
3. Funciones
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 (`MATHEMATICA`, `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`.

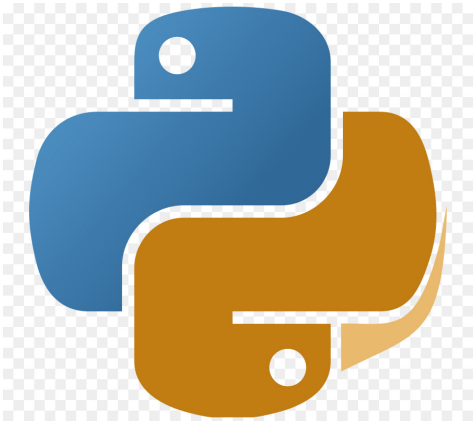
4. **Gratuito**.

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, ...

Índice



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 (`string`) 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**
`ndarray.shape`	Tupla con las dimensiones.
`ndarray.ndim`	Número de dimensiones.
`ndarray.size`	Número de elementos.
`ndarray.itemsize`	Tamaño de uno de los elementos en bytes.
`ndarray.nbytes`	Tamaño total ocupado por los elementos.
`ndarray.dtype`	Tipo de dato de los elementos.
`ndarray.real`	Parte real.
`ndarray.imag`	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
4
```

```
>>> a.nbytes
24
>>> a.dtype
dtype('int32')
>>> 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'' implicito
array(['1', '5', '4.3', '(1+3j)'], dtype='<U11')
>>> 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([b'lin', b'lin', b'33'],  dtype='|S3')
>>>  # ''casting'' explicito
```

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_rpma
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)
>>> # 9 filas y 21 campos
```

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 altenativas 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 un 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, **lanzará una excepción** (IndexError).

Indexación

Posibilidades

Tipo de selección	**Sintaxis**
Un sólo elemento	`array[posicion]`
Varios elementos consecutivos	`array[inicio:fin]`
Elementos en orden cualesquiera	`array[[p1, p2,..., pn]]`
(Novedad respecto a PYTHON Core.)	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 n-dimensional.
- 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]])
```

Ejercicios

Del 1 al 7

`http://python.iaa.es/content/ejercicios-curso-científico`

Otras alternativas para cambiar la "forma" de un array

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

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

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

Función	Descripción
`numpy.hstack(seq)`	Apila los arrays de la secuencia "seq" horizontalmente (agrega columnas).
`numpy.vstack(seq)`	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))
>>> numpy.vstack((a.T, b))
[[ 1.  4.]
 [ 2.  5.]
 [ 3.  6.]
 [ 0.  0.]]
>>> numpy.hstack((a, c))
[[1 2 3 0 1 2 3]
 [4 5 6 4 5 6 7]]
```

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(-1, 4)
>>> a
[-1 0 1 2 3]
>>> b = a/0
>>> b
[-inf  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`.

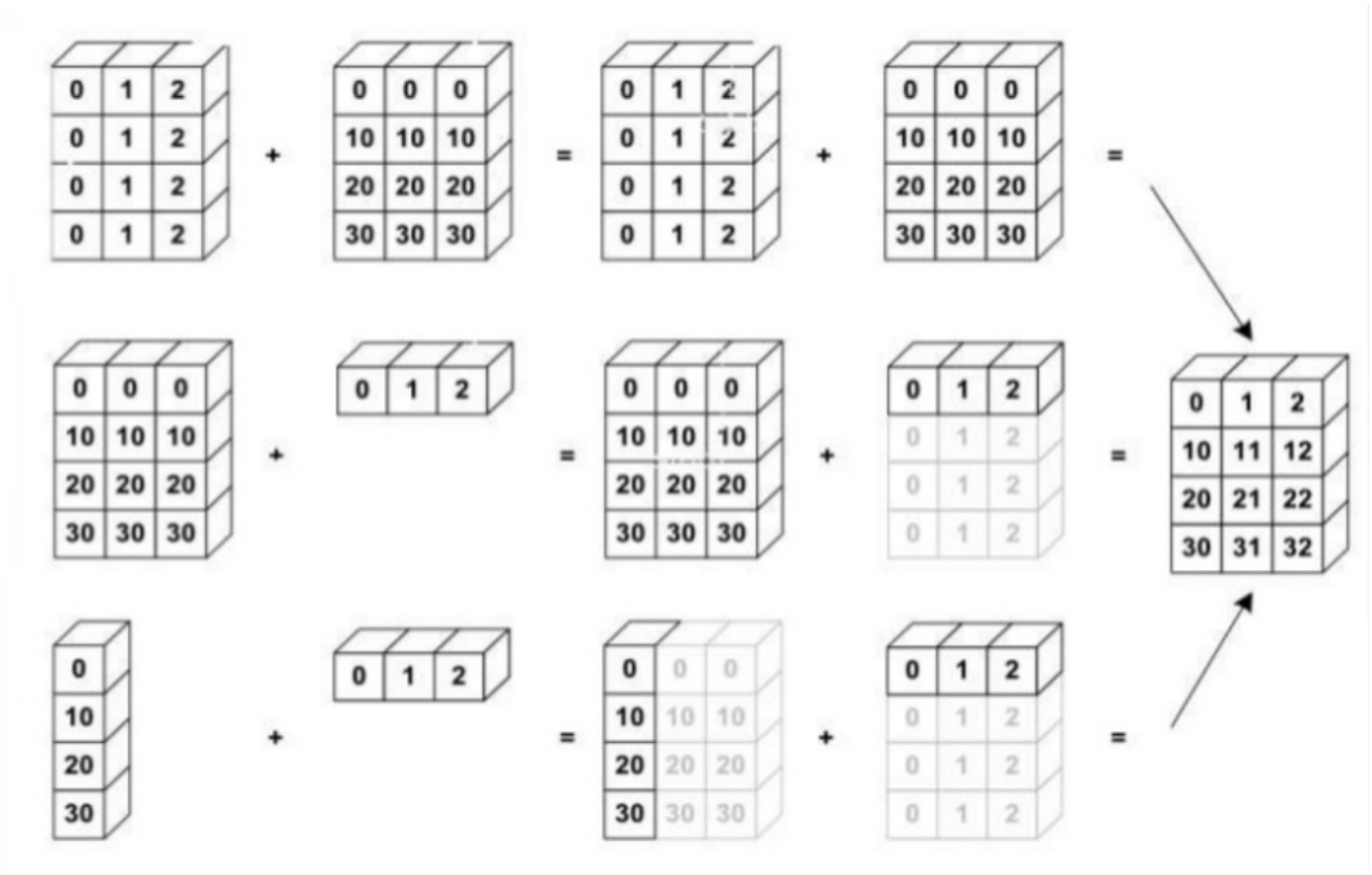
Reglas de broadcasting

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)
>>> a
[0 1 2 3]
>>> a > 2
[False False False  True]
>>> a >= 2
[False False  True  True]
>>> a < 9
[ True  True  True  True]
>>> a <=1
[ True  True False False]
>>> b = numpy.array([0, 1, 3, 3])
>>> a == a
[ True  True  True  True]
>>> a == b
[ True  True False  True]
>>> 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
>>> ab
[False False  True  True]
>>> b = numpy.ones(4, dtype=numpy.bool)
>>> b
[ True  True  True  True]
>>> ab & b
[False False  True  True]
>>> ~ab
[ True  True False False]
>>> ab | b
[ True  True  True  True]
```

Funciones relacionales y lógicas de NUMPY

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

Funciones relacionales y lógicas de NUMPY

Función	Descripción
	Lógicas
`logical_and(x1, x2)`	Determina el valor de x1 AND x2 elemento a elemento.
`logical_or(x1, x2)`	Determina el valor de x1 OR x2 elemento a elemento.
`logical_not(x)`	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])
>>> numpy.allclose(a, b, atol=0.25)
True
>>> numpy.allclose(a, b, atol=0.15)
False
>>> c  = numpy.array([[False, False], [True, True]])
>>> c
[[False False]
 [ True  True]]
>>> numpy.all(c, axis=0)
[False False]
>>> 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])
>>> numpy.array_equal(a, b)
False
>>> numpy.array_equal(a, a)
True
>>> numpy.greater(a, b)
[False False False False]
>>> numpy.greater_equal(a, b)
[ True  True False False]
>>> numpy.less(a, b)
[False False  True  True]
>>> numpy.less_equal(a, b)
[ True  True  True  True]
>>> numpy.equal(a, b)
[ True  True False False]
>>> 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)
>>> a.dtype
int64
>>> a.dtype
dtype('int32')
>>> a.dtype.name
'int32'
```

Otras formas incluyen el uso de funciones NUMPY.

Consulta de tipos de datos

Funciones NUMPY

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

```
>>> a = [1+1j, 1+0j, 4.5, 3, 2, 2j]
>>> numpy.iscomplex(a)
[True, False, False, False, False,  True]
>>> numpy.iscomplexobj(a)
True
>>> numpy.isreal(a)
[ False,  True,  True,  True,  True, False]
>>> numpy.isrealobj(a)
False
>>> numpy.isscalar(a)
False
>>> numpy.isscalar(1)
True
```

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='<U1')
```

¿Qué sucede si intento sumar el número entero 2?

```
>>> a + 2
TypeError:ufunc 'add' did not contain a loop with
signature matching types
dtype('<U3') dtype('<U3') dtype('<U3')'
␣␣␣␣␣
```

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: `nombrearray.astype('typeNumpy')`

`typeNumpy` es cualquiera de dados en la transparencia 18.

Cambio de tipo de dato en un array

Generamos nuevo array

```
>>> a.dtype
dtype('<U1')
>>> a1 = numpy.array(a, dtype='int')
>>> a1.dtype
dtype('int32')
>>> # 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

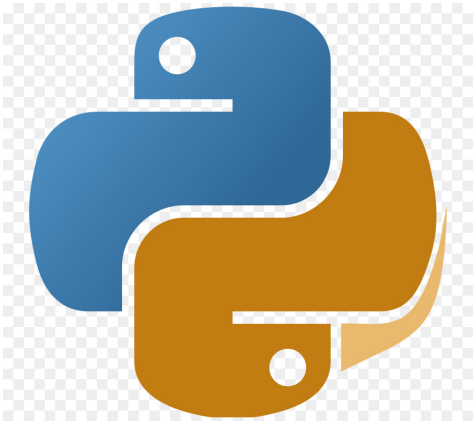
```
>>> a
array(['1', '2', '3', '4'],
      dtype='|S1')
>>> a3 = numpy.complex64(a)
>>> a3
array([ 1.+0.j,  2.+0.j,  3.+0.j,
4.+0.j], dtype=complex64)
>>> a4 = a.astype('int8')
>>> a4.dtype
dtype('int8')
```

Ejercicios

Del 8 al 15

http://python.iaa.es/content/ejercicios-curso-científico

Índice



Métodos de Entrada/Salida

`ndarray.tolist()`	Devuelve el array como una lista.
`ndarray.tofile(fid[,sep,format])`	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.
`ndarray.dump(file)`	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.

El volcado de arrays en fichero es útil cuando se trabaja siempre con el mismo conjunto de datos (aceleramos su lectura), o 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, 0.6000000000000001, 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\9A\99\99\99\99\99\C9?\
>>> a.dump('datos.pick') # volcado de datos a fichero
>>> import pickle
>>> f = open('datos.pick', 'rb')
>>> b = pickle.load(f)
>>> f.close()
>>> b
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
>>> a.dumps()
'\x80\x02cnumpy.core.multiarray\n_reconstruct\nq\x01cnumpy\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`, conjuntop de cadenas que corresponden a valores perdidos.
- `filling_values`, conjunto de valores usados para sustiuir a los valores perdidos.
- `usecols`, (lista/tupla) índices de las columnas que se van a usar (empiezan en 0).
- `names`, None, True, string or 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 s ise quitan espacios al principio y al final de las cadenas de texto que se leen como variables.

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, encoding="utf8")
>>> 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**
`numpy.sort(a, axis=-1)`	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.
`numpy.argsort(a, axis=-1)`	Devuelve un array de posiciones que ordenaría el original de menor a mayor según "axis" dado. Si `axis=None`, transforma a 1D y retorna los índices.

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]  # Atencion 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 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 con los de `a2` si la condición 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
>>> 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
`argmax(a[, axis=None])`	Devuelve los índices de los valores iguales al máximo según el eje "axis". Si `axis=None` buscan en todo el array.
`nanargmax(a[, axis=None])`	Devuelve los índices de los valores iguales al máximo en el eje dado por "axis". Ignora los `NaNs`. Si `axis=None` busca en todo el array.
`argmin(a[, axis=None])`	Devuelve los índices de los valores iguales al mínimo según el eje "axis". Si `axis=None` buscan en todo el array.
`nanargmin(a[, axis=None])`	Devuelve los índices de los valores iguales al mínimo según el eje "axis". Ignora los `NaNs`. Si `axis=None` buscan en todo el array.
`nonzero(a)`	Devuelve una tupla de arrays, una por cada dimensión de *a*, conteniendo los índices de los elementos que no son cero en esa dimensión.
`flatnonzero(a)`	Devuelve los índices de los elementos que no son cero en la versión 1D del array *a*.

Búsqueda en arrays

Función	Descripción
`nonzero(a)`	Devuelve los índices de los elementos que no son cero.
`isnan(a)`	Devuelve un array de booleanos: - *True* en las posiciones donde hay **nan** - *False* en caso contrario.
`isinf(a)`	Devuelve un array de booleanos: - *True* en las posiciones donde hay valores **infinite** - *False* 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)
```

Ejercicios

Del 16 al 21

http://python.iaa.es/content/ejercicios-curso-científico

Operaciones matemáticas

Trigonométricas	**Redondeo**	**Exp & Logs**	**Sum. & Prod. & Diff.**
sin(x)	around(a[, decimals])	exp(x)	prod(a[, axis])
cos(x)	round_(a[, decimals,])	exp1m(x)	sum(a[, axis])
tan(x)	rint(x)	exp2(x)	nansum(a[, axis]))
arcsin(x)	fix(x)	log(x)	cumprod(a[, axis])
arccos(x)	floor(x)	log10(x)	cumsum(a[, axis])
arctan(x)	ceil(x)	log2(x)	gradient(f, *varargs)
hypot(x, y)	trunc(x)	log1p(x)	cross(a, b)
degrees(x)	**Hyperbólicas**	**Miscelánea**	
radians(x)	sinh(x)	sqrt(x)	
deg2rad(x)	cosh(x)	power(x)	
rad2deg(x)	tanh(x)	fbas(x)	
	acrsinh(x)	sign(x)	
	arctan(x)	nan_to_num(x)	

x e y son arrays. Las operaciones se realizan elemento a elemento. Las entradas a funciones trigométricas 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
`numpy.amin(a, axis=None)`	Devuelve un array (o escalar) con el valor mínimo del array a lo largo del eje dado por "axis".
`numpy.amax(a, axis=None)`	Devuelve un array (o escalar) con el valor máximo del array a lo largo del eje dado por "axis".
`numpy.nanmin(a, axis=None)`	Devuelve un array (o escalar) con el valor mínimo del array a lo largo del eje dado por "axis". Ignora los valores `NaN`.
`numpy.nanmax(a, axis=None)`	Devuelve un array (o escalar) con el valor máximo del array a lo largo del eje dado por "axis". Ignora los valores `NaN`.
`numpy.ptp(a, axis=None)`	Devuelve el rango de valores (máximo - mínimo) en el "axis" dado. El nombre de esta función viene del acrónimo "peak to peak".
`numpy.percentile(a, q, axis=None)`	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]]
>>> numpy.amax(a)
8
>>> numpy.amax(a, axis=0)
[5 6 8]
>>> numpy.amax(a, axis=1)
[6 8]
>>> numpy.percentile(a, 25)
2.25
>>> numpy.percentile(a, 25, axis=0)
[ 2.75  3.75  2.75]
>>> numpy.percentile(a, 25, axis=1)
[ 3.   2.5]
>>>numpy.ptp(a)
7
>>> numpy.ptp(a, axis=1)
[5 6]
```

Estadística básica

Ejemplo 2

```
>>> b = numpy.array([[5, numpy.nan, 1], [2, 3, numpy.nan]])
>>> numpy.amin(b)
nan
>>> numpy.amax(b)
nan
>>> numpy.nanmin(b)
1.0
>>> numpy.nanmin(b, axis=0)
[ 2.  3.  1.]
>>> numpy.nanmin(b, axis=1)
[ 1.  2.]
>>> numpy.nanmax(b, axis=1)
[ 5.  3.]
>>> numpy.ptp(b)
nan
>>> numpy.ptp(b, axis=0)
[  3.  nan  nan]
>>> 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
`numpy.average(a, axis=None, weights=None)`	Devuelve un escalar o array con la media "pesada" del array *a* 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 *a* en el eje seleccionado. Si `weights=None` se asume el mismo peso (valor=1) para todos los elementos.
`numpy.mean(a, axis=None, dtype=None)`	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.
`numpy.median(a, axis=None)`	Devuelve un escalar o array con la mediana del array para el eje seleccionado.
`numpy.std(a, axis=None, dtype=None, ..., ddof=0)`	Devuelve un escalar o array con la desviación estándar en el eje seleccionado. `ddof` es el acrónimo de *Delta Degrees of Freedom*. El denominador usado en los cálculos es $N - ddof$, donde N es el número de elementos.
`numpy.var(a, axis=None, dtype=None, ..., ddof=0)`	Devuelve un escalar o array con la varianza de los elementos del array en el eje seleccionado. Misma leyenda que `std` para el resto de parámetros.

Estadística básica

Ejemplo 3

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

Estadística básica

Histogramas

Función	Descripción
`numpy.histogram(a, bins=10, range=None, normed=False, weights=None, density=None)`	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*.
`numpy.histogram2d(x, y, bins=10, range=None, normed=False, weights=None)`	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	Descripción
`numpy.digitize(a, bins, right=False)`	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])
```

Ejercicios

Del 22 al 26

http://python.iaa.es/content/ejercicios-curso-científico

Muestreo aleatorio

Hemos de importar de forma explícita este módulo.

```
import numpy.random
```

Función	Descripción
`rand(d0, d1, ..., dn)`	Devuelve una array de dimensión `(d0, d1, ..., dn)` con números aleatorios en [0, 1).
`randn(d0, d1, ..., dn)`	Muestra de dimensión `(d0, d1, ..., dn)` elementos de una distribución normal estándar.
`randint(low, high=None, size=None)` `high=None, size=None)`	Devuelve "size" `num` enteros en `[start, end)`. `size` puede ser entero a secuencia.
`choice(a, size=1, replace=True, p=None)`	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`.
`shuffle(x)`	Mezcla la secuencia dada por `x`. Cambia `x`. Devuelve None.
`permutation(x)`	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
>>> numpy.random.rand(2, 3)
[[ 0.47817202  0.44355165  0.9863675 ]
 [ 0.72560495  0.11331353  0.21659595]]
>>> numpy.random.randn(4)
[-0.98550123  0.74578984 -2.04972696  0.36567011]
>>> numpy.random.randint(0, 11, size=(2, 3))
[[10  5  0]
 [ 7  1  7]]
>>> a = numpy.arange(5)
>>> numpy.random.choice(a, size=4)
[3 1 2 1]
>>> 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))
>>> numpy.random.permutation(b)
[[6 7 8]
 [3 4 5]
 [0 1 2]]
```

Ejercicios

Del 27 al 30

http://python.iaa.es/content/ejercicios-curso-científico

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

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

Polinomios

Métodos

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

Operaciones entre polinomios

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

Función	Descripción
`polyadd(c1, c2)`	Devuelve la suma dos polinomios.
`polysub(c1, c2)`	Devuelve el polinomio resta de c1 y c2.
`polymul(c1, c2)`	Devuelve el producto de dos polinomios.
`polymulx(c)`	Devuelve el producto de un polinomio por x.
`polydiv(c1, c2)`	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.
`polypow(c, pow)`	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]
>>> npp.polyadd(c1, c2)
[ 0.  3.  3.]
>>> npp.polysub(c1, c2)
[ 0.  1.  3.]
>>> npp.polymul(c1, c2)
[ 0.  0.  2.  3.]
>>> 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**
`polyfit(x, y, deg, ..., full=False, w=None)`	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.

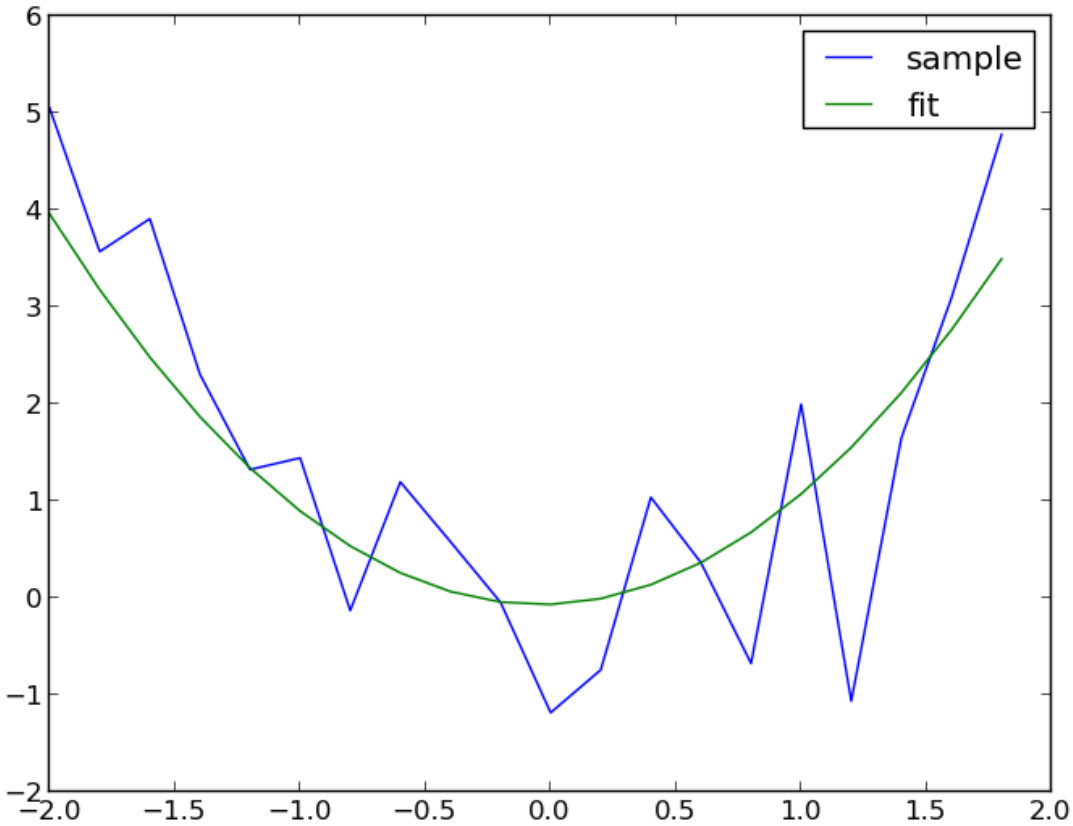
Ajuste

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))
>>> 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 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*

Ejercicios

Del 31 al 36

http://python.iaa.es/content/ejercicios-curso-científico

Arrays con máscaras

Son accesibles a través del paquete `numpy.ma`

1. Es frecuente que en un array haya elementos no válidos (*nan*, *inf*), que no tienen un valor asignado (*missing values*) o que tenga valores que no se desea considerar.
2. En estos casos se pueden usar los arrays con máscara. Se generan por combinación de un array de datos y otro de booleanos (máscara) con las mismas dimensiones.
3. Cuando un elemento del array tiene asociado un valor de máscara igual a **False, quiere decir que ese valor es válido** (no está enmascarado).
4. El paquete numpy.ma se asegura de usar sólo los valores válidos a la hora de realizar operaciones.

Arrays con máscaras

Creación y manejo

Existen varias formas. La más sencilla es a través de la función `array` del módulo.

```
numpy.ma.array(data, dtype=None,
 mask=False, fill_value=None)
```

Donde "`data`" es la secuencia de datos,
"`dtype`" el tipo de dato,
"`mask`" es una secuencia de booleanos que representa la máscara,
"`fill_value`" es el valor con el que se rellena la máscara en caso de ser necesario.

```
>>> import numpy.ma
>>> am = numpy.ma.array([1, 2, 3])
>>> am
masked_array(data = [1 2 3],
             mask = False,
       fill_value = 999999)
```

Arrays con máscaras

Creación y manejo

- Para acceder a los datos, se usa el atributo "`data`". También es posible usar el método "`__array__`" o la función "`getdata`".
- Para acceder a la máscara se usa el atributo "`mask`". Las otras posibilidades pasan por usar las funciones "`getmask`" y "`getmaskarray`" del módulo.

Creación y manejo

Ejemplo 1

```
>>> am = numpy.ma.array([1, 2, 3], mask=[True, False, False])
>>> print am
[-- 2 3]
>>> am.data
array([1, 2, 3])
>>> am.__array__()
array([1, 2, 3])
>>> numpy.ma.getdata(am)
array([1, 2, 3])
>>>
>>> am.mask
[ True False False]
>>> numpy.ma.getmask(am)
[ True False False]
>>> numpy.ma.getmaskarray(am)
[ True False False]
```

Arrays con máscaras

Creación y manejo

Dado

```
>>> am = numpy.ma.array([1, 2, 3], mask=[True, False, False])
>>> am
masked_array(data = [-- 2 3],
             mask = [ True False False],
       fill_value = 999999)
```

Para modificar máscara ...

```
>>> am[1] = numpy.ma.masked  # para un elemento
>>> am
[-- -- 3]
>>> am.mask = [0,1,0]  # para cambiar varios
>>> am
[1 -- 3]
>>> am[1] = -2  # asignando un valor valido se desenmascara
>>> am
[1 -2 3]
```

Arrays con máscaras

Operaciones

```
>>> x = numpy.ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x[0], x[-1]
(1, masked)
>>> x[-1] is numpy.ma.masked
True
```

Para operar con arrays con máscara se hace lo mismo que con arrays normales

```
>>> x = numpy.ma.array([1, 2, 3, 4], mask=[0, 0, 1, 0])
>>> y = numpy.ma.array([3, -1, 2, 5], mask=[0, 0, 1, 1])
>>> x + y
masked_array(data = [4 1 -- --],
             mask = [False False  True  True],
       fill_value = 999999)
```

El módulo numpy.ma gestiona de forma automática las máscaras.

Arrays con máscaras

Funciones numpy.ma

numpy.ma implementa muchas de las funciones vistas para los arrays.

```
>>> y = numpy.ma.array([3, -1, 2, 5], mask=[0, 0, 1, 1])
>>> numpy.ma.max(y)
3
>>> numpy.ma.ptp(y)
4
>>> numpy.ma.std(y)
2.0
>>> numpy.ma.median(y)
1.0
```

Ahora, piense en "arrays con máscara" en N-dimensionales.

¿Ve las posibilidades de uso en su trabajo diario?

Ejercicios

Del 37 al 38

http://python.iaa.es/content/ejercicios-curso-científico

Y por fin...

FIN

P.D.: Nos hemos dejado mucha de la funcionalidad de NUMPY por el camino.

SCIPY es un paquete que se basa en NUMPY, mejorando y ampliando algunas de sus funcionalidades. Parte de lo que falta se verá en ese módulo.