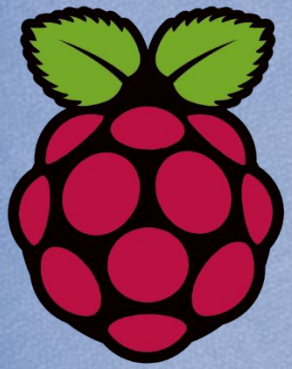
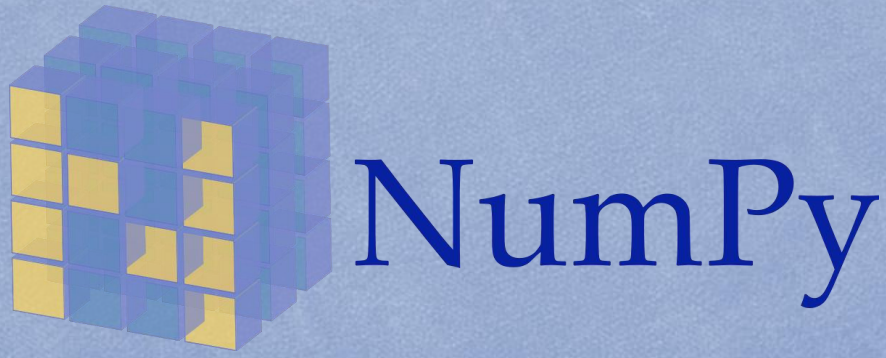


NUMPY

Inicio Rápido

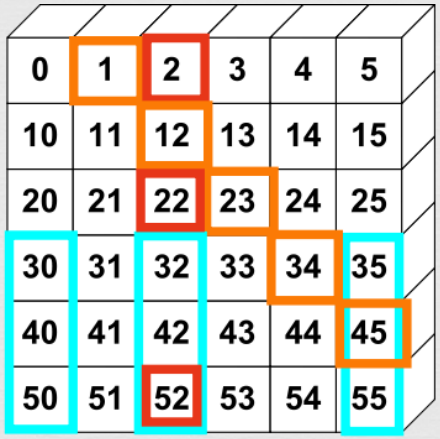
POR. Christian Quispe Canchari



¿Qué es Numpy?

- Numpy es un paquete que provee a Python con arreglos multidimensionales de alta eficiencia y diseñados para cálculo científico.

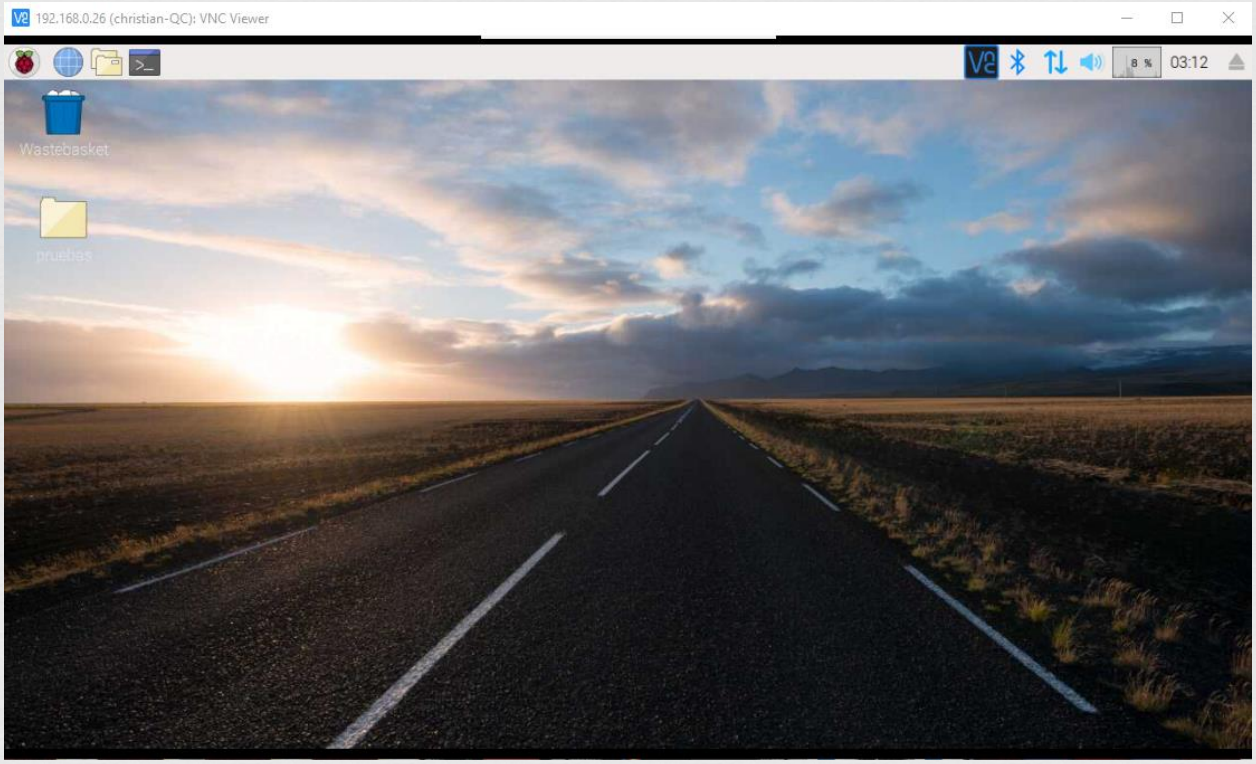
```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45]])
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
array([2,22,52])
```



Temas a tratar

Antes de empezar …

Entramos por VNC a la raspberry y crearemos un directorio llamado prueba para nuestros proyectos en Desktop



NumPy

Prerrequisitos

El requisito fundamental es conocer un poco de Python así como tener la biblioteca Numpy instalado en su Raspberry

Lo Básico

El objeto principan de NumPy es una matriz multidimensional homogenea , compuesta generalmente por números y todos del mismo tipo indexados por una tupla de enteros positivos. En NumPy las dimensiones se llaman ejes.

Creacion de una Matriz

```
>>> import numpy as np
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

“array” transforma secuencias de secuencias en matrices bidimensionales , secuencias de secuencias de secuencias en matrices tridimensionales , y así sucesivamente.

```
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

El tipo de matriz también se puede especificar explícitamente en el momento de la creación:

```
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

A menudo, los elementos de una matriz son originalmente desconocidos, pero se conoce su tamaño. Por lo tanto, NumPy ofrece varias funciones para crear matrices con contenido inicial de marcador de posición.

La función zeros crea una matriz llena de ceros, la función ones crea una matriz llena de unos, y la función empty crea una matriz cuyo contenido inicial es aleatorio y depende del estado de la memoria. Por defecto, la dtype de la matriz creada es float64.

```
>>> np.zeros( (3,4) )
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> np.ones( (2,3,4), dtype=np.int16 )              # dtype can also be specified
array([[[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]],
       [[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]]], dtype=int16)
>>> np.empty( (2,3) )                               # uninitialized, output may vary
array([[  3.73603959e-262,   6.02658058e-154,   6.55490914e-260],
       [  5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
```

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

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> np.arange( 0, 2, 0.3 )                 # it accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

Cuando arange se usa con argumentos de punto flotante, generalmente no es posible predecir el número de elementos obtenidos, debido a la precisión de coma flotante finita. Por esta razón, generalmente es mejor usar la función linspace que recibe como argumento el número de elementos que queremos, en lugar del paso:

```
>>> from numpy import pi
>>> np.linspace( 0, 2, 9 )                 # 9 numbers from 0 to 2
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
>>> x = np.linspace( 0, 2*pi, 100 )        # useful to evaluate function at lots of po
ints
>>> f = np.sin(x)
```

Imprimir matrices

Cuando imprime una matriz, NumPy la muestra de manera similar a las listas anidadas, pero con el siguiente diseño:

•el último eje se imprime de izquierda a derecha,

•el penúltimo está impreso de arriba a abajo,

•el resto también se imprime de arriba a abajo, con cada porción separada de la siguiente por una línea vacía.

Las matrices unidimensionales se imprimen como filas, bidimensionales como matrices y tridimensionales como listas de matrices.

```
>>> a = np.arange(6)                         # 1d array
>>> print(a)
[0 1 2 3 4 5]
>>>
>>> b = np.arange(12).reshape(4,3)           # 2d array
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>>
>>> c = np.arange(24).reshape(2,3,4)         # 3d array
>>> print(c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]
 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Si una matriz es demasiado grande para imprimirse, NumPy omite automáticamente la parte central de la matriz y solo imprime las esquinas:

```
>>> print(np.arange(10000))
[   0    1    2 ..., 9997 9998 9999]
>>>
>>> print(np.arange(10000).reshape(100,100))
[[   0    1    2 ...,   97   98   99]
 [ 100  101  102 ...,  197  198  199]
 [ 200  201  202 ...,  297  298  299]
 ...,
 [9700 9701 9702 ..., 9797 9798 9799]
 [9800 9801 9802 ..., 9897 9898 9899]
 [9900 9901 9902 ..., 9997 9998 9999]]
```

Para desactivar este comportamiento y forzar a NumPy a imprimir toda la matriz, puede cambiar las opciones de impresión utilizando set_printoptions.

```
>>> np.set_printoptions(threshold=np.nan)
```

Operaciones Básicas

Los operadores aritméticos en matrices se aplican de forma *electrónica* . Se crea una nueva matriz y se completa con el resultado.

```
>>> a = np.array( [20,30,40,50] )
>>> b = np.arange( 4 )
>>> b
array([0, 1, 2, 3])
>>> c = a-b
>>> c
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> 10*np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
>>> a<35
array([ True, True, False, False])
```

A diferencia de muchos lenguajes de matriz, el operador del producto *opera en forma de elemento en matrices NumPy. El producto de matriz se puede realizar utilizando el @operador (en python> = 3.5) o la función dot o método:

```
>>> A = np.array( [[1,1],
...               [0,1]] )
>>> B = np.array( [[2,0],
...               [3,4]] )
>>> A * B                          # elementwise product
array([[2, 0],
       [0, 4]])
>>> A @ B                          # matrix product
array([[5, 4],
       [3, 4]])
>>> A.dot(B)                       # another matrix product
array([[5, 4],
       [3, 4]])
```

Algunas operaciones, como +=y *=, actúan en su lugar para modificar una matriz existente en lugar de crear una nueva.

```
>>>
>>> a = np.ones((2,3), dtype=int)
>>> b = np.random.random((2,3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[ 3.417022  ,  3.72032449,  3.00011437],
       [ 3.30233257,  3.14675589,  3.09233859]])
>>> a += b                  # b is not automatically converted to integer type
Traceback (most recent call last):
  ...
TypeError: Cannot cast ufunc add output from dtype('float64') to dtype('int64') with c
asting rule 'same_kind'
```

Al operar con matrices de diferentes tipos, el tipo de matriz resultante corresponde a la más general o precisa (un comportamiento conocido como upcasting).

```
>>> a = np.ones(3, dtype=np.int32)
>>> b = np.linspace(0,pi,3)
>>> b.dtype.name
'float64'
>>> c = a+b
>>> c
array([ 1.        ,  2.57079633,  4.14159265])
>>> c.dtype.name
'float64'
>>> d = np.exp(c*1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```

Muchas operaciones únicas, como calcular la suma de todos los elementos en la matriz, se implementan como métodos de la ndarrayclase.

```
>>> a = np.random.random((2,3))
>>> a
array([[ 0.18626021,  0.34556073,  0.39676747],
       [ 0.53881673,  0.41919451,  0.6852195 ]])
>>> a.sum()
2.5718191614547998
>>> a.min()
0.1862602113776709
>>> a.max()
0.6852195003967595
```

Por defecto, estas operaciones se aplican a la matriz como si fuera una lista de números, independientemente de su forma. Sin embargo, al especificar el axis parámetro, puede aplicar una operación a lo largo del eje especificado de una matriz:

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> b.sum(axis=0)                           # sum of each column
array([12, 15, 18, 21])
>>>
>>> b.min(axis=1)                           # min of each row
array([0, 4, 8])
>>>
>>> b.cumsum(axis=1)                        # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

Funciones universales

NumPy proporciona funciones matemáticas familiares como sin, cos y exp. En NumPy, estos se llaman "funciones universales" (ufunc). Dentro de NumPy, estas funciones operan como elementos en una matriz, produciendo una matriz como salida.

```
>>> B = np.arange(3)
>>> B
array([0, 1, 2])
>>> np.exp(B)
array([ 1.        ,  2.71828183,  7.3890561 ])
>>> np.sqrt(B)
array([ 0.        ,  1.        ,  1.41421356])
>>> C = np.array([2., -1., 4.])
>>> np.add(B, C)
array([ 2.,  0.,  6.])
```

Indexación, corte e iteración

Las matrices **unidimensionales** se pueden indexar, dividir e iterar, al igual que las listas y otras secuencias de Python.

```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
>>> a[:6:2] = -1000    # equivalent to a[0:6:2] = -1000; from start to position 6, exc
lusive, set every 2nd element to -1000
>>> a
array([-1000,     1, -1000,    27, -1000,   125,   216,   343,   512,   729])
>>> a[ : :-1]                                 # reversed a
array([  729,   512,   343,   216,   125, -1000,    27, -1000,     1, -1000])
>>> for i in a:
...     print(i**(1/3.))
...
nan
1.0
nan
3.0
nan
5.0
6.0
7.0
8.0
9.0
```

Las matrices **multidimensionales** pueden tener un índice por eje. Estos índices se dan en una tupla separada por comas:

```
>>> def f(x,y):
...     return 10*x+y
...
>>> b = np.fromfunction(f,(5,4),dtype=int)
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
>>> b[0:5, 1]                       # each row in the second column of b
array([ 1, 11, 21, 31, 41])
>>> b[ : ,1]                        # equivalent to the previous example
array([ 1, 11, 21, 31, 41])
>>> b[1:3, : ]                      # each column in the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Cuando se proporcionan menos índices que el número de ejes, los índices faltantes se consideran cortes completos:

```
>>> b[-1]                                # the last row. Equivalent to b[-1,:]
array([40, 41, 42, 43])
```

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

Los puntos (...) representan tantos puntos como sea necesario para producir una tupla de indexación completa. Por ejemplo, si xes una matriz con 5 ejes, entonces

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

```
>>> c = np.array( [[[  0,  1,  2],               # a 3D array (two stacked 2D arrays)
...                 [ 10, 12, 13]],
...                [[100,101,102],
...                 [110,112,113]]])
>>> c.shape
(2, 2, 3)
>>> c[1,...]                                   # same as c[1,:,:] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])
>>> c[...,2]                                   # same as c[:,:,2]
array([[  2,  13],
       [102, 113]])
```

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

```
>>> for row in b:
...     print(row)
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

Sin embargo, si uno quiere realizar una operación en cada elemento de la matriz, uno puede usar el flatatributo que es un iterador sobre todos los elementos de la matriz:

```
>>> for element in b.flat:
...     print(element)
...
0
1
2
3
10
11
12
13
20
21
22
23
30
31
32
33
40
41
42
43
```

Manipulación de formas

Cambiar la forma de una matriz

Una matriz tiene una forma dada por la cantidad de elementos a lo largo de cada eje:

```
>>> a = np.floor(10*np.random.random((3,4)))
>>> a
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
>>> a.shape
(3, 4)
```

La forma de una matriz se puede cambiar con varios comandos. Tenga en cuenta que los tres comandos siguientes devuelven una matriz modificada, pero no cambian la matriz original:

```
>>> a.ravel()  # returns the array, flattened
array([ 2.,  8.,  0.,  6.,  4.,  5.,  1.,  1.,  8.,  9.,  3.,  6.])
>>> a.reshape(6,2)  # returns the array with a modified shape
array([[ 2.,  8.],
       [ 0.,  6.],
       [ 4.,  5.],
       [ 1.,  1.],
       [ 8.,  9.],
       [ 3.,  6.]])
>>> a.T  # returns the array, transposed
array([[ 2.,  4.,  8.],
       [ 8.,  5.,  9.],
       [ 0.,  1.,  3.],
       [ 6.,  1.,  6.]])
>>> a.T.shape
(4, 3)
>>> a.shape
(3, 4)
```

El orden de los elementos en la matriz resultante de ravel () es normalmente "estilo C", es decir, el índice de la derecha "cambia más rápido", por lo que el elemento después de a [0,0] es un [0,1] . Si la matriz se reforma a alguna otra forma, de nuevo la matriz se trata como "C-style". NumPy normalmente crea matrices almacenadas en este orden, por lo que ravel () generalmente no necesitará copiar su argumento, pero si la matriz se creó tomando segmentos de otra matriz o creada con opciones inusuales, puede que necesite copiarse. Las funciones ravel () y reshape () también pueden instruirse, utilizando un argumento opcional, para usar arreglos tipo FORTRAN, en los cuales el índice más a la izquierda cambia más rápido.

La reshapefunción devuelve su argumento con una forma modificada, mientras que el ndarray.resizemétodo modifica la matriz en sí:

```
>>> a
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
>>> a.resize((2,6))
>>> a
array([[ 2.,  8.,  0.,  6.,  4.,  5.],
       [ 1.,  1.,  8.,  9.,  3.,  6.]])
```

Si una dimensión se da como -1 en una operación de remodelación, las otras dimensiones se calculan automáticamente:

```
>>> a.reshape(3,-1)
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
```

Apilando diferentes matrices

Varias matrices se pueden apilar juntas a lo largo de diferentes ejes:

```
>>> a = np.floor(10*np.random.random((2,2)))
>>> a
array([[ 8.,  8.],
       [ 0.,  0.]])
>>> b = np.floor(10*np.random.random((2,2)))
>>> b
array([[ 1.,  8.],
       [ 0.,  4.]])
>>> np.vstack((a,b))
array([[ 8.,  8.],
       [ 0.,  0.],
       [ 1.,  8.],
       [ 0.,  4.]])
>>> np.hstack((a,b))
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
```

La función column_stack apila matrices 1D como columnas en una matriz 2D. Es equivalente hstacksolo para matrices 2D:

```
>>> from numpy import newaxis
>>> np.column_stack((a,b))     # with 2D arrays
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
>>> a = np.array([4.,2.])
>>> b = np.array([3.,8.])
>>> np.column_stack((a,b))     # returns a 2D array
array([[ 4., 3.],
       [ 2., 8.]])
>>> np.hstack((a,b))           # the result is different
array([ 4., 2., 3., 8.])
>>> a[:,newaxis]               # this allows to have a 2D columns vector
array([[ 4.],
       [ 2.]])
>>> np.column_stack((a[:,newaxis],b[:,newaxis]))
array([[ 4.,  3.],
       [ 2.,  8.]])
>>> np.hstack((a[:,newaxis],b[:,newaxis]))   # the result is the same
array([[ 4.,  3.],
       [ 2.,  8.]])
```

Por otro lado, la función row_stackes equivalente a vstack cualquier matriz de entrada. En general, para arreglos de más de dos dimensiones, se hstackapila a lo largo de sus segundos ejes, se vstackapila a lo largo de sus primeros ejes y concatenate permite un argumento opcional que proporciona el número del eje a lo largo del cual debe ocurrir la concatenación.

Nota

En casos complejos, r_y c_son útiles para crear matrices apilando números a lo largo de un eje. Permiten el uso de literales de rango (":")

```
>>> np.r_[1:4,0,4]
array([1, 2, 3, 0, 4])
```

Cuando se utiliza con matrices como argumentos, r_y c_son similares a vstack, y hstacken su comportamiento por defecto, pero permitir un argumento opcional que indica el número del eje largo de la cual concatenar.

Dividir una matriz en varias más pequeñas

Utilizando hsplit, puede dividir una matriz a lo largo de su eje horizontal, ya sea especificando el número de matrices con la misma forma a devolver, o especificando las columnas después de las cuales debería ocurrir la división:

```
>>> a = np.floor(10*np.random.random((2,12)))
>>> a
array([[ 9.,  5.,  6.,  3.,  6.,  8.,  0.,  7.,  9.,  7.,  2.,  7.],
       [ 1.,  4.,  9.,  2.,  2.,  1.,  0.,  6.,  2.,  2.,  4.,  0.]])
>>> np.hsplit(a,3)   # Split a into 3
[array([[ 9.,  5.,  6.,  3.],
       [ 1.,  4.,  9.,  2.]]), array([[ 6.,  8.,  0.,  7.],
       [ 2.,  1.,  0.,  6.]]), array([[ 9.,  7.,  2.,  7.],
       [ 2.,  2.,  4.,  0.]])]
>>> np.hsplit(a,(3,4))   # Split a after the third and the fourth column
[array([[ 9.,  5.,  6.],
       [ 1.,  4.,  9.]]), array([[ 3.],
       [ 2.]]), array([[ 6.,  8.,  0.,  7.,  9.,  7.,  2.,  7.],
       [ 2.,  1.,  0.,  6.,  2.,  2.,  4.,  0.]])]
```

vsplitse divide a lo largo del eje vertical y array_splitpermite especificar a qué eje dividir.

Copias y Vistas

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

No copiar nada

Las asignaciones simples no hacen ninguna copia de los objetos de la matriz o de sus datos.

```
>>> a = np.arange(12)
>>> b = a            # no new object is created
>>> b is a           # a and b are two names for the same ndarray object
True
>>> b.shape = 3,4    # changes the shape of a
>>> a.shape
(3, 4)
```

Python pasa objetos mutables como referencias, por lo que las llamadas a funciones no hacen ninguna copia.

```
>>> def f(x):
...     print(id(x))
...
>>> id(a)                           # id is a unique identifier of an object
148293216
>>> f(a)
148293216
```

Vista o copia superficial

Diferentes objetos de matriz pueden compartir los mismos datos. El viewmétodo crea un nuevo objeto de matriz que analiza los mismos datos.

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a                        # c is a view of the data owned by a
True
>>> c.flags.owndata
False
>>>
>>> c.shape = 2,6                      # a's shape doesn't change
>>> a.shape
(3, 4)
>>> c[0,4] = 1234                      # a's data changes
>>> a
array([[   0,    1,    2,    3],
       [1234,    5,    6,    7],
       [   8,    9,   10,   11]])
```

Cortar una matriz devuelve una vista de ella:

```
>>> s = a[ : , 1:3]     # spaces added for clarity; could also be written "s = a[:,1:3]"
>>> s[:] = 10           # s[:] is a view of s. Note the difference between s=10 and s[:]=10
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

Copia profunda

El copy método hace una copia completa de la matriz y sus datos.

```
>>> d = a.copy()                          # a new array object with new data is created
>>> d is a
False
>>> d.base is a                           # d doesn't share anything with a
False
>>> d[0,0] = 9999
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

Menos básico

Normas de Broadcasting

La radiodifusión permite que las funciones universales se manejen de manera significativa con las entradas que no tienen exactamente la misma forma.
La primera regla de radiodifusión es que si todas las matrices de entrada no tienen el mismo número de dimensiones, un "1" se antepondrá repetidamente a las formas de las matrices más pequeñas hasta que todas las matrices tengan el mismo número de dimensiones.
La segunda regla de transmisión garantiza que las matrices con un tamaño de 1 a lo largo de una dimensión particular actúen como si tuvieran el tamaño de la matriz con la forma más grande a lo largo de esa dimensión. Se supone que el valor del elemento de la matriz es el mismo a lo largo de esa dimensión para la matriz de "difusión".
Después de la aplicación de las reglas de transmisión, los tamaños de todas las matrices deben coincidir. Se pueden encontrar más detalles en Broadcasting .

Trucos fáciles de indexación e índice

NumPy ofrece más funciones de indexación que las secuencias normales de Python. Además de indexar por enteros y sectores, como vimos antes, las matrices pueden indexarse mediante matrices de enteros y matrices de booleanos.

Indexación con matrices de índices

```
>>> a = np.arange(12)**2                       # the first 12 square numbers
>>> i = np.array( [ 1,1,3,8,5 ] )              # an array of indices
>>> a[i]                                       # the elements of a at the positions i
array([ 1,  1,  9, 64, 25])
>>>
>>> j = np.array( [ [ 3, 4], [ 9, 7 ] ] )      # a bidimensional array of indices
>>> a[j]                                       # the same shape as j
array([[ 9, 16],
       [81, 49]])
```

Cuando la matriz indexada aes multidimensional, una única matriz de índices se refiere a la primera dimensión de a. El siguiente ejemplo muestra este comportamiento al convertir una imagen de etiquetas en una imagen en color utilizando una paleta.

```
>>> palette = np.array( [ [0,0,0],                # black
...                      [255,0,0],              # red
...                      [0,255,0],              # green
...                      [0,0,255],              # blue
...                      [255,255,255] ] )       # white
>>> image = np.array( [ [ 0, 1, 2, 0 ],          # each value corresponds to a color in the palette
...                     [ 0, 3, 4, 0 ]  ] )
>>> palette[image]                            # the (2,4,3) color image
array([[[  0,   0,   0],
        [255,   0,   0],
        [  0, 255,   0],
        [  0,   0,   0]],
       [[  0,   0,   0],
        [  0,   0, 255],
        [255, 255, 255],
        [  0,   0,   0]]])
```

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

```
>>> a = np.arange(12).reshape(3,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> i = np.array( [ [0,1],                       # indices for the first dim of a
...                 [1,2] ] )
>>> j = np.array( [ [2,1],                       # indices for the second dim
...                 [3,3] ] )
>>>
>>> a[i,j]                                     # i and j must have equal shape
array([[ 2,  5],
       [ 7, 11]])
>>>
>>> a[i,2]
array([[ 2,  6],
       [ 6, 10]])
>>>
>>> a[:,j]                                     # i.e., a[ : , j]
array([[[ 2,  1],
        [ 3,  3]],
       [[ 6,  5],
        [ 7,  7]],
       [[10,  9],
        [11, 11]]])
```

Naturalmente, podemos poner iy jen una secuencia (digamos una lista) y luego hacer la indexación con la lista.

```
>>> l = [i,j]
>>> a[l]                                    # equivalent to a[i,j]
array([[ 2,  5],
       [ 7, 11]])
```

Sin embargo, no podemos hacer esto poniendo iy jdentro de una matriz, porque esta matriz se interpretará como la indexación de la primera dimensión de a.

```
>>> s = np.array( [i,j] )
>>> a[s]                                    # not what we want
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: index (3) out of range (0<=index<=2) in dimension 0
>>>
>>> a[tuple(s)]                             # same as a[i,j]
array([[ 2,  5],
       [ 7, 11]])
```

Otro uso común de la indexación con matrices es la búsqueda del valor máximo de series dependientes del tiempo:

```
>>> time = np.linspace(20, 145, 5)                 # time scale
>>> data = np.sin(np.arange(20)).reshape(5,4)      # 4 time-dependent series
>>> time
array([  20.  ,   51.25,   82.5 ,  113.75,  145.  ])
>>> data
array([[ 0.        ,  0.84147098,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ,  0.6569866 ],
       [ 0.98935825,  0.41211849, -0.54402111, -0.99999021],
       [-0.53657292,  0.42016704,  0.99060736,  0.65028784],
       [-0.28790332, -0.96139749, -0.75098725,  0.14987721]])
>>>
>>> ind = data.argmax(axis=0)                 # index of the maxima for each series
>>> ind
array([2, 0, 3, 1])
>>>
>>> time_max = time[ind]                      # times corresponding to the maxima
>>>
>>> data_max = data[ind, range(data.shape[1])] # => data[ind[0],0], data[ind[1],1]...
>>>
>>> time_max
array([  82.5 ,   20.  ,  113.75,   51.25])
>>> data_max
array([ 0.98935825,  0.84147098,  0.99060736,  0.6569866 ])
>>>
>>> np.all(data_max == data.max(axis=0))
True
```

También puede usar la indexación con matrices como un destino para asignar a:

```
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a[[1,3,4]] = 0
>>> a
array([0, 0, 2, 0, 0])
```

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

```
>>> a = np.arange(5)
>>> a[[0,0,2]]=[1,2,3]
>>> a
array([2, 1, 3, 3, 4])
```

Esto es bastante razonable, pero ten cuidado si quieres usar la +=construcción de Python , ya que puede no hacer lo que esperas:

```
>>> a = np.arange(5)
>>> a[[0,0,2]]+=1
>>> a
array([1, 1, 3, 3, 4])
```

Aunque 0 aparece dos veces en la lista de índices, el elemento 0º solo se incrementa una vez. Esto se debe a que Python requiere que "a + = 1" sea equivalente a "a = a + 1"

Indexación con matrices booleanas

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

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

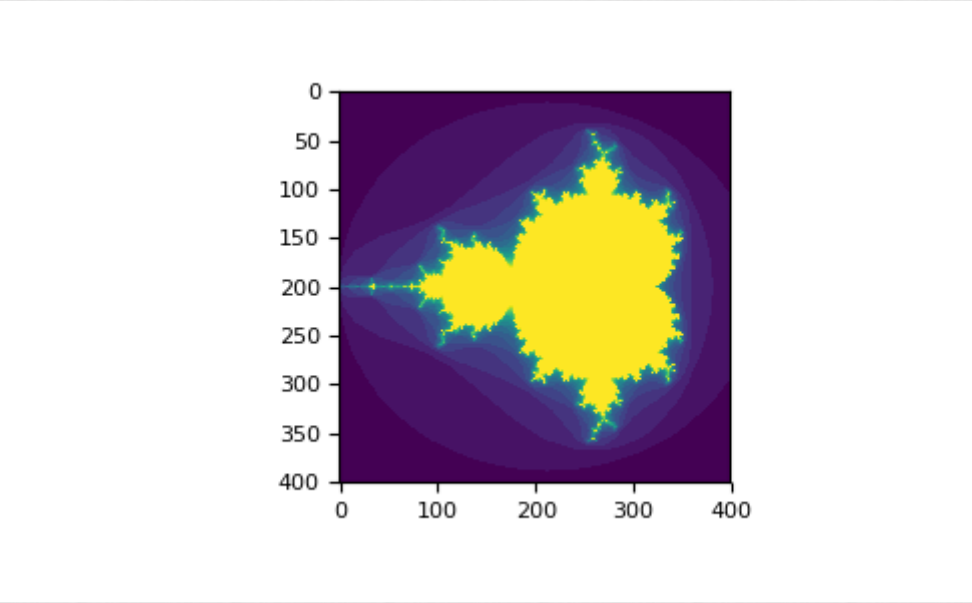
```
>>> a = np.arange(12).reshape(3,4)
>>> b = a > 4
>>> b                                    # b is a boolean with a's shape
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]])
>>> a[b]                                 # 1d array with the selected elements
array([ 5,  6,  7,  8,  9, 10, 11])
```

Esta propiedad puede ser muy útil en asignaciones:

```
>>> a[b] = 0                                # All elements of 'a' higher than 4 become 0
>>> a
array([[0, 1, 2, 3],
       [4, 0, 0, 0],
       [0, 0, 0, 0]])
```

Puede ver el siguiente ejemplo para ver cómo usar la indexación booleana para generar una imagen del [conjunto de Mandelbrot](#) :

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> def mandelbrot( h,w, maxit=20 ):
...     """Returns an image of the Mandelbrot fractal of size (h,w)."""
...     y,x = np.ogrid[ -1.4:1.4:h*1j, -2:0.8:w*1j ]
...     c = x+y*1j
...     z = c
...     divtime = maxit + np.zeros(z.shape, dtype=int)
...
...     for i in range(maxit):
...         z = z**2 + c
...         diverge = z*np.conj(z) > 2**2            # who is diverging
...         div_now = diverge & (divtime==maxit)  # who is diverging now
...         divtime[div_now] = i                  # note when
...         z[diverge] = 2                        # avoid diverging too much
...
...     return divtime
>>> plt.imshow(mandelbrot(400,400))
>>> plt.show()
```



La segunda forma de indexar con booleanos es más similar a la indexación entera; para cada dimensión de la matriz damos una matriz booleana 1D seleccionando las divisiones que queremos:

```
>>> a = np.arange(12).reshape(3,4)
>>> b1 = np.array([False,True,True])            # first dim selection
>>> b2 = np.array([True,False,True,False])      # second dim selection
>>>
>>> a[b1,:]                                   # selecting rows
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[b1]                                     # same thing
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[:,b2]                                   # selecting columns
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
>>>
>>> a[b1,b2]                                  # a weird thing to do
array([ 4, 10])
```

Tenga en cuenta que la longitud de la matriz booleana 1D debe coincidir con la longitud de la dimensión (o eje) que desea cortar. En el ejemplo anterior, b1tiene longitud 3 (el número de filas en a), y b2(de longitud 4) es adecuado para indexar el 2º eje (columnas) de a.

La función ix_ ()

La ix_función se puede usar para combinar diferentes vectores a fin de obtener el resultado para cada n-uplet. Por ejemplo, si desea calcular todos los a + b * c para todos los trillizos tomados de cada uno de los vectores a, byc:

```
>>> a = np.array([2,3,4,5])
>>> b = np.array([8,5,4])
>>> c = np.array([5,4,6,8,3])
>>> ax,bx,cx = np.ix_(a,b,c)
>>> ax
array([[[2]],
       [[3]],
       [[4]],
       [[5]]])
>>> bx
array([[[8],
        [5],
        [4]]])
>>> cx
array([[[5, 4, 6, 8, 3]]])
>>> ax.shape, bx.shape, cx.shape
((4, 1, 1), (1, 3, 1), (1, 1, 5))
>>> result = ax+bx*cx
>>> result
array([[[42, 34, 50, 66, 26],
        [27, 22, 32, 42, 17],
        [22, 18, 26, 34, 14]],
       [[43, 35, 51, 67, 27],
        [28, 23, 33, 43, 18],
        [23, 19, 27, 35, 15]],
       [[44, 36, 52, 68, 28],
        [29, 24, 34, 44, 19],
        [24, 20, 28, 36, 16]],
       [[45, 37, 53, 69, 29],
        [30, 25, 35, 45, 20],
        [25, 21, 29, 37, 17]]])
>>> result[3,2,4]
17
>>> a[3]+b[2]*c[4]
17
```

También puede implementar la reducción de la siguiente manera:

```
>>> def ufunc_reduce(ufct, *vectors):
...     vs = np.ix_(*vectors)
...     r = ufct.identity
...     for v in vs:
...         r = ufct(r,v)
...     return r
```

y luego usarlo como:

```
>>> ufunc_reduce(np.add,a,b,c)
array([[[15, 14, 16, 18, 13],
        [12, 11, 13, 15, 10],
        [11, 10, 12, 14,  9]],
       [[16, 15, 17, 19, 14],
        [13, 12, 14, 16, 11],
        [12, 11, 13, 15, 10]],
       [[17, 16, 18, 20, 15],
        [14, 13, 15, 17, 12],
        [13, 12, 14, 16, 11]],
       [[18, 17, 19, 21, 16],
        [15, 14, 16, 18, 13],
        [14, 13, 15, 17, 12]]])
```

La ventaja de esta versión de reducir en comparación con el ufunc.reduce normal es que utiliza las reglas de difusión para evitar crear una matriz de argumentos del tamaño de la salida multiplicada por el número de vectores.