Arreglos

Las estructuras de datos que hemos visto hasta ahora (listas, tuplas, diccionarios, conjuntos) permiten manipular datos de manera muy flexible. Combinándolas y anidándolas, es posible organizar información de manera estructurada para representar sistemas del mundo real.

En muchas aplicaciones de Ingeniería, por otra parte, más importante que la organización de los datos es la capacidad de hacer muchas operaciones a la vez sobre grandes conjuntos de datos numéricos de manera eficiente. Algunos ejemplos de problemas que requieren manipular grandes secuencias de números son: la predicción del clima, la construcción de edificios, y el análisis de indicadores financieros entre muchos otros.

La estructura de datos que sirve para almacenar estas grandes secuencias de números (generalmente de tipo `float`) es el **arreglo**.

Los arreglos tienen algunas similitudes con las listas:

- los elementos tienen un orden y se pueden acceder mediante su posición,
- los elementos se pueden recorrer usando un ciclo `for`.

Sin embargo, también tienen algunas restricciones:

- todos los elementos del arreglo deben tener el mismo tipo,
- en general, el tamaño del arreglo es fijo (no van creciendo dinámicamente como las listas),
- se ocupan principalmente para almacenar datos numéricos.

A la vez, los arreglos tienen muchas ventajas por sobre las listas, que iremos descubriendo a medida que avancemos en la materia.

Los arreglos son los equivalentes en programación de las **matrices** y **vectores** de las matemáticas. Precisamente, una gran motivación para usar arreglos es que hay mucha teoría detrás de ellos que puede ser usada en el diseño de algoritmos para resolver problemas verdaderamente interesantes.

Crear arreglos

El módulo que provee las estructuras de datos y las funciones para trabajar con arreglos se llama **NumPy**, y no viene incluído con Python, por lo que hay que instalarlo por separado.

Descargue el instalador apropiado para su versión de Python desde la página de descargas de NumPy. Para ver qué versión de Python tiene instalada, vea la primera línea que aparece al abrir una consola.

Para usar las funciones provistas por NumPy, debemos importarlas al principio del programa:

```
from numpy import array
```

Como estaremos usando frecuentemente muchas funciones de este módulo, conviene importarlas todas de una vez usando la siguiente sentencia:

```
from numpy import *
```

(Si no recuerda cómo usar el `import`, puede repasar la materia sobre módulos).

El tipo de datos de los arreglos se llama `array`. Para crear un arreglo nuevo, se puede usar la función `array` pasándole como parámetro la lista de valores que deseamos agregar al arreglo:

```
>>> a = array([6, 1, 3, 9, 8])
>>> a
array([6, 1, 3, 9, 8])
```

Todos los elementos del arreglo tienen exactamente el mismo tipo. Para crear un arreglo de números reales, basta con que uno de los valores lo sea:

```
>>> b = array([6.0, 1, 3, 9, 8])
>>> b
array([ 6.,  1.,  3.,  9.,  8.])
```

Otra opción es convertir el arreglo a otro tipo usando el método `astype`:

```
>>> a
array([6, 1, 3, 9, 8])
>>> a.astype(float)
array([ 6.,  1.,  3.,  9.,  8.])
>>> a.astype(complex)
array([ 6.+0.j,  1.+0.j,  3.+0.j,  9.+0.j,  8.+0.j])
```

Hay muchas formas de arreglos que aparecen a menudo en la práctica, por lo que existen funciones especiales para crearlos:

- `zeros(n)` crea un arreglo de `n` ceros;
- `ones(n)` crea un arreglo de `n` unos;
- `arange(a, b, c)` crea un arreglo de forma similar a la función `range`, con las diferencias que `a`, `b` y `c` pueden ser reales, y que el resultado es un arreglo y no una lista;
- `linspace(a, b, n)` crea un arreglo de `n` valores equiespaciados entre `a` y `b`.

```
>>> zeros(6)
array([ 0.,  0.,  0.,  0.,  0.,  0.])

>>> ones(5)
array([ 1.,  1.,  1.,  1.,  1.])

>>> arange(3.0, 9.0)
array([ 3.,  4.,  5.,  6.,  7.,  8.])

>>> linspace(1, 2, 5)
array([ 1.  ,  1.25,  1.5 ,  1.75,  2.  ])
```

Operaciones con arreglos

Las limitaciones que tienen los arreglos respecto de las listas son compensadas por la cantidad de operaciones convenientes que permiten realizar sobre ellos.

Las operaciones aritméticas entre arreglos se aplican elemento a elemento:

```
>>> a = array([55, 21, 19, 11,  9])
>>> b = array([12, -9,  0, 22, -9])

# sumar los dos arreglos elemento a elemento
>>> a + b
array([67, 12, 19, 33,  0])

# multiplicar elemento a elemento
>>> a * b
array([ 660, -189,    0,  242,  -81])

# restar elemento a elemento
>>> a - b
array([ 43,  30,  19, -11,  18])
```

Las operaciones entre un arreglo y un valor simple funcionan aplicando la operación a todos los elementos del arreglo, usando el valor simple como operando todas las veces:

```
>>> a
array([55, 21, 19, 11,  9])

# multiplicar por 0.1 todos los elementos
>>> 0.1 * a
array([ 5.5,  2.1,  1.9,  1.1,  0.9])

# restar 9.0 a todos los elementos
>>> a - 9.0
array([ 46.,  12.,  10.,   2.,   0.])
```

Note que si quisiéramos hacer estas operaciones usando listas, necesitaríamos usar un ciclo para hacer las operaciones elemento a elemento.

Las operaciones relacionales también se aplican elemento a elemento, y retornan un arreglo de valores booleanos:

```
>>> a = array([5.1, 2.4, 3.8, 3.9])
>>> b = array([4.2, 8.7, 3.9, 0.3])
>>> c = array([5, 2, 4, 4]) + array([1, 4, -2, -1]) / 10.0

>>> a < b
array([False,  True,  True, False], dtype=bool)

>>> a == c
array([ True,  True,  True,  True], dtype=bool)
```

Para reducir el arreglo de booleanos a un único valor, se puede usar las funciones `any` y `all`. `any` retorna `True` si al menos uno de los elementos es verdadero, mientras que `all` retorna `True` sólo si todos lo son (en inglés, *any* signfica «alguno», y *all* significa «todos»):

```
>>> any(a < b)
True
>>> any(a == b)
False
>>> all(a == c)
True
```

Funciones sobre arreglos

NumPy provee muchas funciones matemáticas que también operan elemento a elemento. Por ejemplo, podemos obtener el seno de 9 valores equiespaciados entre 0 y $\pi/2$ con una sola llamada a la función `sin`:

```
>>> from numpy import linspace, pi, sin

>>> x = linspace(0, pi/2, 9)
>>> x
array([ 0.        ,  0.19634954,  0.39269908,
        0.58904862,  0.78539816,  0.9817477 ,
        1.17809725,  1.37444679,  1.57079633])

>>> sin(x)
array([ 0.        ,  0.19509032,  0.38268343,
        0.55557023,  0.70710678,  0.83146961,
        0.92387953,  0.98078528,  1.        ])
```

Como puede ver, los valores obtenidos crecen desde 0 hasta 1, que es justamente como se comporta la función seno en el intervalo $[0, \pi/2]$.

Aquí también se hace evidente otra de las ventajas de los arreglos: al mostrarlos en la consola o al imprimirlos, los valores aparecen perfectamente alineados. Con las listas, esto no ocurre:

```
>>> list(sin(x))
[0.0, 0.19509032201612825, 0.38268343236508978, 0.5555702330
1960218, 0.70710678118654746, 0.83146961230254524, 0.9238795
3251128674, 0.98078528040323043, 1.0]
```

Arreglos aleatorios

El módulo NumPy contiene a su vez otros módulos que proveen funcionalidad adicional a los arreglos y funciones básicos.

El módulo `numpy.random` provee funciones para crear **números aleatorios** (es decir, generados al azar), de las cuales la más usada es la función `random`, que entrega un arreglo de números al azar distribuidos uniformemente entre 0 y 1:

```
>>> from numpy.random import random

>>> random(3)
array([ 0.53077263,  0.22039319,  0.81268786])
>>> random(3)
array([ 0.07405763,  0.04083838,  0.72962968])
>>> random(3)
array([ 0.51886706,  0.46220545,  0.95818726])
```

Obtener elementos de un arreglo

Cada elemento del arreglo tiene un índice, al igual que en las listas. El primer elemento tiene índice 0. Los elementos también pueden numerarse desde el final hasta el principio usando índices negativos. El último elemento tiene índice −1:

```
>>> a = array([6.2, -2.3, 3.4, 4.7, 9.8])

>>> a[0]
6.2
>>> a[1]
-2.3
>>> a[-2]
4.7
>>> a[3]
4.7
```

Una seccion del arreglo puede ser obtenida usando el operador de rebanado `a[i:j]`. Los índices `i` y `j` indican el rango de valores que serán entregados:

```
>>> a
array([ 6.2, -2.3,  3.4,  4.7,  9.8])
>>> a[1:4]
array([-2.3,  3.4,  4.7])
>>> a[2:-2]
array([ 3.4])
```

Si el primer índice es omitido, el rebanado comienza desde el principio del arreglo. Si el segundo índice es omitido, el rebanado termina al final del arreglo:

```
>>> a[:2]
array([ 6.2, -2.3])
>>> a[2:]
array([ 3.4,  4.7,  9.8])
```

Un tercer índice puede indicar cada cuántos elementos serán incluídos en el resultado:

```
>>> a = linspace(0, 1, 9)
>>> a
array([ 0.   ,  0.125,  0.25 ,  0.375,  0.5  ,  0.625,  0.75 ,  0.875,  1.
])
>>> a[1:7:2]
array([ 0.125,  0.375,  0.625])
>>> a[::3]
array([ 0.   ,  0.375,  0.75 ])
>>> a[-2::-2]
array([ 0.875,  0.625,  0.375,  0.125])
>>> a[::-1]
array([ 1.   ,  0.875,  0.75 ,  0.625,  0.5  ,  0.375,  0.25 ,  0.125,  0.
])
```

Una manera simple de recordar cómo funciona el rebanado es considerar que los índices no se refieren a los elementos, sino a los espacios entre los elementos:

```
>>> b = array([17.41, 2.19, 10.99, -2.29, 3.86, 11.10])
>>> b[2:5]
array([ 10.99,  -2.29,   3.86])
>>> b[:5]
array([ 17.41,   2.19,  10.99,  -2.29,   3.86])
>>> b[1:1]
array([], dtype=float64)
>>> b[1:5:2]
array([ 2.19, -2.29])
```

Algunos métodos convenientes

Los arreglos proveen algunos métodos útiles que conviene conocer.

Los métodos `min` y `max`, entregan respectivamente el mínimo y el máximo de los elementos del arreglo:

```
>>> a = array([4.1, 2.7, 8.4, pi, -2.5, 3, 5.2])
>>> a.min()
-2.5
>>> a.max()
8.4000000000000004
```

Los métodos `argmin` y `argmax` entregan respectivamente la posición del mínimo y del máximo:

```
>>> a.argmin()
4
>>> a.argmax()
2
```

Los métodos `sum` y `prod` entregan respectivamente la suma y el producto de los elementos:

```
>>> a.sum()
24.041592653589795
>>> a.prod()
-11393.086289208301
```

Arreglos bidimensionales

Los **arreglos bidimensionales** son tablas de valores. Cada elemento de un arreglo bidimensional está simultáneamente en una fila y en una columna.

En matemáticas, a los arreglos bidimensionales se les llama matrices, y son muy utilizados en problemas de Ingeniería.

En un arreglo bidimensional, cada elemento tiene una posición que se identifica mediante dos índices: el de su fila y el de su columna.

Crear arreglos bidimensionales

Los arreglos bidimensionales también son provistos por NumPy, por lo que debemos comenzar importando las funciones de este módulo:

```
from numpy import *
```

Al igual que los arreglos de una dimensión, los arreglos bidimensionales también pueden ser creados usando la función `array`, pero pasando como argumentos una lista con las filas de la matriz:

```
a = array([[5.1, 7.4, 3.2, 9.9],
           [1.9, 6.8, 4.1, 2.3],
           [2.9, 6.4, 4.3, 1.4]])
```

Todas las filas deben ser del mismo largo, o si no ocurre un error de valor:

```
>>> array([[1], [2, 3]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: setting an array element with a sequence.
```

Los arreglos tienen un atributo llamado `shape`, que es una tupla con los tamaños de cada dimensión. En el ejemplo, `a` es un arreglo de dos dimensiones que tiene tres filas y cuatro columnas:

```
>>> a.shape
(3, 4)
```

Los arreglos también tienen otro atributo llamado `size` que indica cuántos elementos tiene el arreglo:

```
>>> a.size
12
```

Por supuesto, el valor de `a.size` siempre es el producto de los elementos de `a.shape`.

Hay que tener cuidado con la función `len`, ya que no retorna el tamaño del arreglo, sino su cantidad de filas:

```
>>> len(a)
3
```

Las funciones `zeros` y `ones` también sirven para crear arreglos bidimensionales. En vez de pasarles como argumento un entero, hay que entregarles una tupla con las cantidades de filas y columnas que tendrá la matriz:

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

>>> ones((2, 5))
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

Lo mismo se cumple para muchas otras funciones que crean arreglos; por ejemplom la función `random`:

```
>>> from numpy.random import random
>>> random((5, 2))
array([[ 0.80177393,  0.46951148],
       [ 0.37728842,  0.72704627],
       [ 0.56237317,  0.3491332 ],
       [ 0.35710483,  0.44033758],
       [ 0.04107107,  0.47408363]])
```

Operaciones con arreglos bidimensionales

Al igual que los arreglos de una dimensión, las operaciones sobre las matrices se aplican término a término:

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

>>> a + 2
array([[7, 3, 6],
       [2, 5, 4]])

>>> a ** b
array([[25,  1,  0],
      [ 0,  1,  2]])
```

Cuando dos matrices aparecen en una operación, ambas deben tener exactamente la misma forma:

```
>>> a = array([[5, 1, 4],
...            [0, 3, 2]])
>>> b = array([[ 2,  3],
...            [-1,  1],
...            [ 0,  1]])
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

Obtener elementos de un arreglo bidimensional

Para obtener un elemento de un arreglo, debe indicarse los índices de su fila `i` y su columna `j` mediante la sintaxis `a[i, j]`:

```
>>> a = array([[ 3.21,  5.33,  4.67,  6.41],
               [ 9.54,  0.30,  2.14,  6.57],
               [ 5.62,  0.54,  0.71,  2.56],
               [ 8.19,  2.12,  6.28,  8.76],
               [ 8.72,  1.47,  0.77,  8.78]])
>>> a[1, 2]
2.14

>>> a[4, 3]
8.78

>>> a[-1, -1]
8.78

>>> a[0, -1]
6.41
```

También se puede obtener secciones rectangulares del arreglo usando el operador de rebanado con los índices:

```
>>> a[2:3, 1:4]
array([[ 0.54,  0.71,  2.56]])

>>> a[1:4, 0:4]
array([[ 9.54,  0.3 ,  2.14,  6.57],
       [ 5.62,  0.54,  0.71,  2.56],
       [ 8.19,  2.12,  6.28,  8.76]])

>>> a[1:3, 2]
array([ 2.14,  0.71])

>>> a[0:4:2, 3:0:-1]
array([[ 6.41,  4.67,  5.33],
       [ 2.56,  0.71,  0.54]])

>>> a[::4, ::3]
array([[ 3.21,  6.41],
       [ 8.72,  8.78]])
```

Para obtener una fila completa, hay que indicar el índice de la fila, y poner `:` en el de las columnas (significa «desde el principio hasta el final»). Lo mismo para las columnas:

```
>>> a[2, :]
array([ 5.62,  0.54,  0.71,  2.56])

>>> a[:, 3]
array([ 6.41,  6.57,  2.56,  8.76,  8.78])
```

Note que el número de dimensiones es igual a la cantidad de rebanados que hay en los índices:

```
>>> a[2, 3]      # valor escalar (arreglo de cero dimensiones)
2.56

>>> a[2:3, 3]    # arreglo de una dimensión de 1 elemento
array([ 2.56])

>>> a[2:3, 3:4]  # arreglo de dos dimensiones de 1 x 1
array([[ 2.56]])
```

Otras operaciones

La **trasposicion** consiste en cambiar las filas por las columnas y viceversa. Para trasponer un arreglo, se usa el método `transpose`:

```
>>> a
array([[ 3.21,  5.33,  4.67,  6.41],
       [ 9.54,  0.3 ,  2.14,  6.57],
       [ 5.62,  0.54,  0.71,  2.56]])

>>> a.transpose()
array([[ 3.21,  9.54,  5.62],
       [ 5.33,  0.3 ,  0.54],
       [ 4.67,  2.14,  0.71],
       [ 6.41,  6.57,  2.56]])
```

El método `reshape` entrega un arreglo que tiene los mismos elementos pero otra forma. El parámetro de `reshape` es una tupla indicando la nueva forma del arreglo:

```
>>> a = arange(12)
>>> a
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

>>> a.reshape((4, 3))
array([[ 0, 1, 2],
       [ 3, 4, 5],
       [ 6, 7, 8],
       [ 9, 10, 11]])

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

La función `diag` aplicada a un arreglo bidimensional entrega la diagonal principal de la matriz (es decir, todos los elementos de la forma `a[i, i]`):

```
>>> a
array([[ 3.21,  5.33,  4.67,  6.41],
       [ 9.54,  0.3 ,  2.14,  6.57],
       [ 5.62,  0.54,  0.71,  2.56]])

>>> diag(a)
array([ 3.21,  0.3 ,  0.71])
```

Además, `diag` recibe un segundo parámetro opcional para indicar otra diagonal que se desee obtener. Las diagonales sobre la principal son positivas, y las que están bajo son negativas:

```
>>> diag(a, 2)
array([ 4.67,  6.57])
>>> diag(a, -1)
array([ 9.54,  0.54])
```

La misma función `diag` también cumple el rol inverso: al recibir un arreglo de una dimensión, retorna un arreglo bidimensional que tiene los elementos del parámetro en la diagonal:

```
>>> diag(arange(5))
array([[0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 0, 3, 0],
       [0, 0, 0, 0, 4]])
```

Reducciones por fila y por columna

Algunas operaciones pueden aplicarse tanto al arreglo completo como a todas las filas o a todas las columnas.

Por ejemplo, `a.sum()` entrega la suma de todos los elementos del arreglo. Además, se le puede pasar un parámetro para hacer que la operación se haga por filas o por columnas:

```
>>> a = array([[ 4.3,  2.9,  9.1,  0.1,  2. ],
...            [ 8. ,  4.5,  6.4,  6. ,  4.3],
...            [ 7.8,  3.1,  3.4,  7.8,  8.4],
...            [ 1.2,  1.5,  9. ,  6.3,  6.8],
...            [ 7.6,  9.2,  3.3,  0.9,  8.6],
...            [ 5.3,  6.7,  4.6,  5.3,  1.2],
...            [ 4.6,  9.1,  1.5,  3. ,  0.6]])
>>> a.sum()
174.4
>>> a.sum(0)
array([ 38.8,  37. ,  37.3,  29.4,  31.9])
>>> a.sum(1)
array([ 18.4,  29.2,  30.5,  24.8,  29.6,  23.1,  18.8])
```

El parámetro indica a lo largo de qué dimensión se hará la suma. El `0` significa «sumar a lo largo de las filas». Pero hay que tener cuidado, ¡por que lo que se obtiene son las sumas de las columnas! Del mismo modo, `1` significa «a lo largo de las columnas, y lo que se obtiene es el arreglo con las sumas de cada fila.

Las operaciones `a.min()` y `a.max()` funcionan del mismo modo:

```
>>> a.min()
0.1
>>> a.min(0)
array([ 1.2,  1.5,  1.5,  0.1,  0.6])
>>> a.min(1)
array([ 0.1,  4.3,  3.1,  1.2,  0.9,  1.2,  0.6])
```

`a.argmin()` y `a.argmax()` también:

```
>>> a.argmin(0)
array([3, 3, 6, 0, 6])
>>> a.argmin(1)
array([3, 4, 1, 0, 3, 4, 4])
```

Productos entre arreglos

Recordemos que **vector** es sinónimo de arreglo de una dimensión, y **matriz** es sinónimo de arreglo de dos dimensiones.

Producto interno (vector-vector)

El **producto interno** entre dos vectores es la suma de los productos entre elementos correspondientes:

El producto interno entre dos vectores se obtiene usando la función `dot` provista por NumPy:

```
>>> a = array([-2.8 , -0.88,  2.76,  1.3 ,  4.43])
>>> b = array([ 0.25, -1.58,  1.32, -0.34, -4.22])
>>> dot(a, b)
-14.803
```

El producto interno es una operación muy común. Por ejemplo, suele usarse para calcular totales:

```
>>> precios = array([200, 100, 500, 400, 400, 150])
>>> cantidades = array([1, 0, 0, 2, 1, 0])
>>> total_a_pagar = dot(precios, cantidades)
>>> total_a_pagar
1400
```

También se usa para calcular promedios ponderados:

```
>>> notas = array([45, 98, 32])
>>> ponderaciones = array([30, 30, 40]) / 100.
>>> nota_final = dot(notas, ponderaciones)
>>> nota_final
55.7
```

Producto matriz-vector

El **producto matriz-vector** es el vector de los productos internos El producto matriz-vector puede ser visto simplemente como varios productos internos calculados de una sola vez.

Esta operación también es obtenida usando la función `dot` entre las filas de la matriz y el vector:

El producto matriz-vector puede ser visto simplemente como varios productos internos calculados de una sola vez.

Esta operación también es obtenida usando la función `dot`:

```
>>> a = array([[-0.6,  4.8, -1.2],
               [-2. , -3.6, -2.1],
               [ 1.7,  4.9,  0. ]])
>>> x = array([-0.6, -2. ,  1.7])
>>> dot(a, x)
array([-11.28,   4.83, -10.82])
```

Producto matriz-matriz

El **producto matriz-matriz** es la matriz de los productos internos entre las filas de la primera matriz y las columnas de la segunda:

Esta operación también es obtenida usando la función `dot`:

```
>>> a = array([[ 2,  8],
               [-3,  7],
               [-8, -5]])
>>> b array([[-3, -5, -6, -3],
            [-9, -2,  3, -3]])
>>> dot(a, b)
array([[-78, -26,  12, -30],
       [-54,   1,  39, -12],
       [ 69,  50,  33,  39]])
```

La multiplicación de matrices puede ser vista como varios productos matriz-vector (usando como vectores todas las filas de la segunda matriz), calculados de una sola vez.

En resumen, al usar la función `dot`, la estructura del resultado depende de cuáles son los parámetros pasados:

```
dot(vector, vector) → número
dot(matriz, vector) → vector
dot(matriz, matriz) → matriz
```