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
```