Python para Análisis de Datos

Módulo 03

Numpy

Vectorización

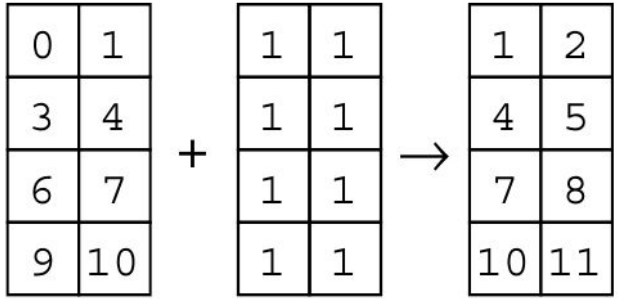
Numpy ofrece la posibilidad de realizar operaciones sobre todos los elementos de los arrays sin la necesidad de recurrir a bucles. Esto es lo que se conoce como **vectorización**. Internamente, Numpy utiliza rutinas muy optimizadas escritas en C, que implementan estas operaciones.

Esta es una de las características esenciales de Numpy. Libera al programador de los detalles de implementación y resulta en código muy compacto y legible.

Operaciones aritméticas

Los operadores aritméticos de Python también están implementados para operar con arrays. Estos operadores trabajan elemento a elemento y devuelven como resultado un nuevo array.

Además de los operadores a los que estamos acostumbrados, también existen funciones correspondientes. Por ejemplo, para la suma existe el operador **+** y la función **`add.`** De modo que `a+b` es equivalente a `np.add(a,b)`



```
x
```

```
array([1, 2, 3, 4])
```

```
y
```

```
array([ 1,  4, 12, 20])
```

```
x/y
```

```
array([1.  , 0.5 , 0.25, 0.2 ])
```

```
a
```

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

```
b
```

```
array([[ 0, 20, 40, 60],
       [10, 30, 50, 70]])
```

```
a + b
```

```
array([[ 0, 21, 42, 63],
       [14, 35, 56, 77]])
```

Incluso se puede operar un array con un número y va a realizar la operación sobre todos los elementos.

```
a
```

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

```
10*a
```

```
array([ 0, 10, 20, 30, 40, 50])
```

```
a**2 +10*a +0.1
```

```
array([ 0.1, 11.1, 24.1, 39.1, 56.1, 75.1])
```

Funciones matemáticas

Así mismo existen numerosas funciones para realizar operaciones matemáticas elemento a elemento.

- Funciones trigonométricas (`sin, cos, tan, arcsin, tanh,` etc.)
- Logarítmicas (`log, log2, log10,` etc)
- Exponenciales (`exp, exp2, expm1,` etc.)
- Potenciales (`square, sqrt, cbrt, reciprocal,` etc.)
- Redondeo de decimales (`round, rint, trunc, floor, ceil,` etc.)
- Y muchas otras.

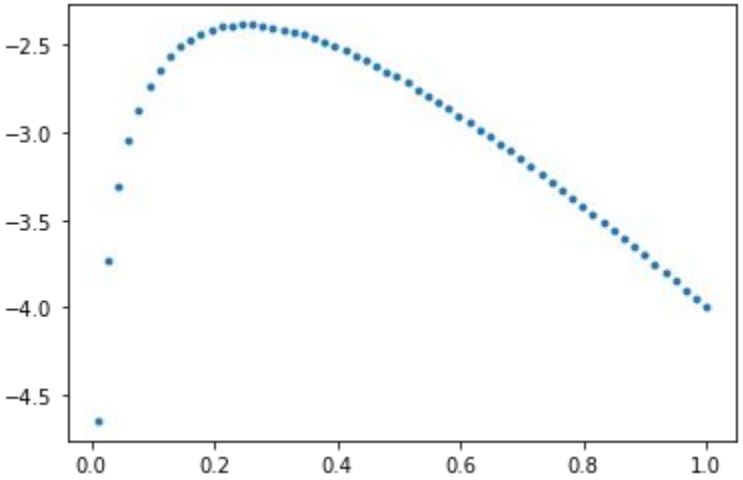
Ejemplo

Creamos el array `x` como un rango lineal equiespaciado de `n` números entre `x0` y `x1`.

Luego, creamos el array `y1` como el logaritmo de `x` menos cuatro veces `x`. Graficamos el resultado.

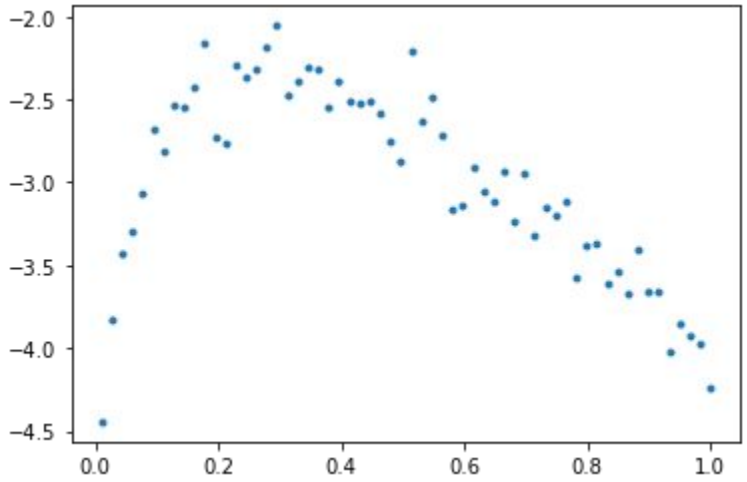
```
x0 = 0.01
x1 = 1
n = 60
x = np.linspace(x0,x1,n)
```

```
y1 = np.log(x) - 4*x
```



Luego, creamos el array y2 como y1 más la suma de "ruido estadístico" simulado con números aleatorios de distribución gaussiana.

```
y2 = y1 + 0.15*np.random.randn(n)
```



Broadcasting

Se conoce como **broadcasting** a la capacidad de operar entre arrays de distinta forma o dimensión (con ciertas restricciones). En ciertas condiciones, Numpy es capaz de promover la forma del array más pequeño a la del array más grande.

10	20	30
40	50	60
70	80	90
100	110	120

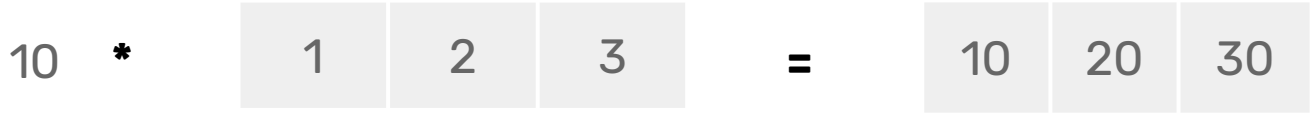
+

1	2	3

=

11	22	33
41	52	63
71	82	93
101	112	123

Cuando operamos un número con un array también se está aplicando broadcasting



La regla es que para que dos dimensiones sean compatibles ambas deben ser iguales o una de ellas debe ser 1 (la dimensión que se promueve).

Operaciones lógicas

Así como es posible usar los operadores aritméticos con arrays, también es posible usar los operadores de comparación. En este caso, se devuelve un array de booleanos que son el resultado de aplicar las comparaciones elemento a elemento.

```
a
```

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

```
b
```

```
array([[ 1, 10,  6,  4],
       [ 3,  9,  1,  8],
       [ 0,  6, 10,  4]])
```

```
a < b
```

```
array([[ True,  True,  True,  True],
       [False,  True, False,  True],
       [False, False, False, False]])
```

Esto es de gran utilidad porque, como vimos, se puede usar un array de booleanos para seleccionar elementos. La siguiente expresión devuelve los elementos de **a** que son menores que el correspondiente elemento de **b.**

```
a[a < b]
```

```
array([0, 1, 2, 3, 5, 7])
```

Nota: las reglas de broadcasting también se aplican a los operadores de comparación.

Pero para operar con arrays de booleanos no es posible usar los operadores **`and`**, **`or`** y **`not`** tal cual como se escriben en Python.

```
bool_1 = np.array([True, False])
bool_2 = np.array([True, True])
bool_1 and bool_2
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
```

Para poder realizar operaciones lógicas entre arrays de booleanos hay que usar ciertas funciones o los operadores especiales correspondientes.

Función	Operador
`np.logical_and`	`&`
`np.logical_or`	`|`
`np.logical_not`	`~`
`np.logicar_xor`	`^`

Así podemos combinar varias condiciones en un solo filtro. La siguiente expresión devuelve los elementos de **a** que son menores que el correspondiente elemento de **b,** en donde el elemento de **b** es par.

```
a[(a < b) & (b%2==0)]
```

```
array([1, 2, 3, 7])
```

¡Importante!

Los operadores lógicos y de comparación no tienen un orden de precedencia en expresiones con arrays sino que se realizan en el orden en el que son escritos. Para que la expresión funcione correctamente hay que encerrar cada parte entre paréntesis. La siguiente expresión (que no encierra las comparaciones entre paréntesis) falla.

```
a[a < b & b%2==0]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call l
ast)
```

Revisión

1. Repasar el concepto de vectorización.
2. Realizar operaciones con varios arrays.
3. Aplicar funciones a distintos arrays y combinarlas con operaciones.
4. Operar con arrays de distinta dimensión para familiarizarse con el concepto de broadcasting.
5. Realizar operaciones lógicas y de comparación entre arrays.
6. Usar condiciones complejas para seleccionar elementos de un array.

¡Muchas gracias!

¡Sigamos trabajando!