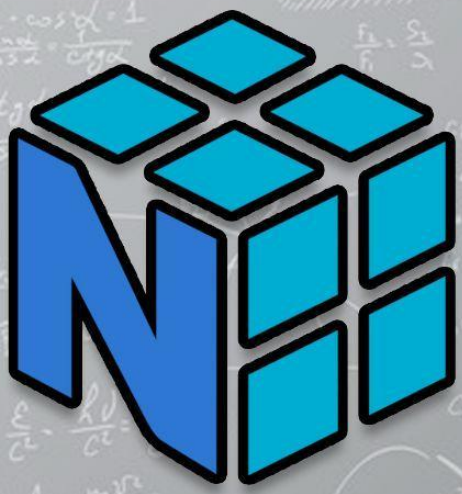
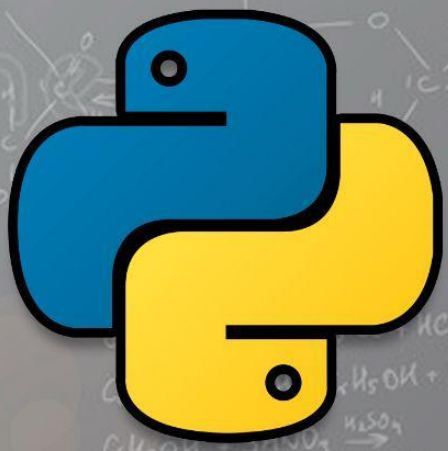
GUÍA BÁSICA DE



NumPy

LA LIBRERÍA MÁS IMPORTANTE DE



AGORA

ÍNDICE DE CONTENIDOS

1. INTRODUCCIÓN A NUMPY

NumPy (*Numerical Python*) es una de las librerías más importantes (por no decir la más importante) de Python.

Fundamentalmente es una librería de **funciones matemáticas**, centrada en el manejo de **arrays**, pero realmente ofrece muchas más utilidades, como iremos viendo más adelante.

1.1. ¿QUÉ ES UN ARRAY?

Probablemente una de las palabras que más veces se va a repetir a lo largo de esta guía es “array”, así que creo que una buena forma de empezar es definiendo exactamente qué es un array:

En programación, un array (también conocido como vector o matriz) es un tipo de dato estructurado que almacena datos del mismo tipo. Un ejemplo de array, vector o matriz unidimensional de 10 elementos sería el siguiente (los números que aparecen encima representan el índice asignado a cada posición del array):

0	1	2	3	4	5	6	7	8	9
1	3	5	7	9	8	6	4	2	0

1.2. ¿POR QUÉ SURGE NUMPY?

En Python, no existe un tipo de dato específico para trabajar con arrays, sino que necesitamos utilizar listas para ello. El problema es que las listas son lentas de procesar, lo que reduce significativamente el rendimiento de nuestro programa.

Buscando resolver este problema, Travis Olisphant creó NumPy en 2005. Actualmente el proyecto es de código abierto y todo el mundo puede utilizarlo líbremente.

El principal objetivo de NumPy es ofrecer un tipo de datos para los arrays, que puede ser procesado 50 veces más rápido que las listas nativas de Python.

El tipo de dato para los arrays que nos ofrece NumPy recibe el nombre de `ndarray`, y también incluye un gran número de funciones para hacer muy sencillo el trabajo con este tipo de datos.

¿Por qué son tan rápidos los arrays de NumPy en comparación con las listas de Python?

Los arrays de NumPy se almacenan en un lugar continuo de la memoria del sistema, a diferencia de las listas, por lo que los procesos pueden acceder a ellos y manipularlos de forma mucho más eficiente.

1.3. INSTALACIÓN Y PRIMEROS PASOS

Para instalar Python utilizando PIP (lo más recomendable), simplemente debemos escribir lo siguiente en una terminal del sistema:

```
pip install numpy
```

Una vez instalado, si queremos utilizarlo en nuestro código, debemos importarlo:

```
import numpy
```

Habitualmente (es un consenso establecido en la comunidad), para trabajar con NumPy utilizamos el alias `np`:

```
import numpy as np
```

Una vez hecho esto, ya estamos en disposición de meternos a hablar en detalle de NumPy. Pero antes, una última cosa. A lo largo de toda la guía encontraréis dos tipos de recuadros: los recuadros grises son código Python, mientras que los recuadros azules son la salida por pantalla de la ejecución del código del recuadro gris que tienen encima. Por ejemplo, si queremos conocer la versión de NumPy que tenemos instalada, podemos hacer lo siguiente:

```
import numpy as np

print(np.__version__)
```

```
1.23.3
```

2. LOS ARRAYS DE NUMPY

En este segundo capítulo de la guía vamos a aprender todo lo necesario para trabajar con los arrays de NumPy.

2.1. CREACIÓN DE ARRAYS

Hemos comentado anteriormente que el tipo de dato que nos ofrece NumPy para trabajar con arrays se denomina `ndarray`. Para crear un objeto de este tipo, usamos la función `array()`:

```
import numpy as np

mi_array = np.array([1, 2, 3])

# Imprimimos por pantalla el array que hemos creado:

print(mi_array)

# Imprimimos por pantalla el tipo de dato, para comprobar que
# efectivamente es un dato de tipo ndarray:

print(type(mi_array))
```

```
[1 2 3]
<class 'numpy.ndarray'>
```

Destacar que, como argumento de entrada, la función `array()` ha recibido una lista de Python. Realmente no es necesario que le pasemos una lista, podemos pasarle cualquier tipo de dato que almacene colecciones de datos, como por ejemplo una tupla.

Una de las características más interesantes de los arrays de NumPy es que pueden tener todas las dimensiones que queramos. El array con la dimensión más pequeña que podemos crear es con **dimensión cero** (también conocido como *scalars*, que son los elementos que hay dentro del array). Por ejemplo:

```
import numpy as np

mi_array = np.array(11)

print(mi_array)

print(type(mi_array))
```

```
11
<class 'numpy.ndarray'>
```

Un array de **una dimensión** está formado a su vez por varios arrays de dimensión cero (esta es la dimensión más frecuente con la que solemos trabajar):

```
import numpy as np

mi_array = np.array([1, 3, 5, 7])

print(mi_array)
```

```
[1 3 5 7]
```

Un array de **dos dimensiones** está formado por varios arrays de una dimensión y habitualmente es lo que llamamos matrices:

```
import numpy as np

mi_array = np.array([[1, 3, 5, 7], [2, 4, 6, 8]])

print(mi_array)
```

```
[[1 3 5 7]
 [2 4 6 8]]
```

Destacar que NumPy nos ofrece todo un submódulo con funciones dedicadas a trabajar con este tipo de matrices de dos dimensiones (el módulo se llama `numpy.mat`, y hablaremos de él más adelante).

Un array de **tres dimensiones** es un array cuyos elementos son matrices de dos dimensiones:

```
import numpy as np

mi_array = np.array([[[1, 2, 3],[4, 5, 6]], [[1, 2, 3],[4, 5, 6]]])

print(mi_array)
```

```
[[[1 2 3]
  [4 5 6]]

 [[1 2 3]
  [4 5 6]]]
```

NumPy nos permite crear de forma muy sencilla arrays con todas las dimensiones que queramos, a través del argumento de entrada `ndim`:

```
import numpy as np

mi_array = np.array([1, 2, 3], ndim=5)

print(mi_array)
```

```
[[[[[1 2 3]]]]]
```

Si en un momento dado, queremos conocer cuántas dimensiones tiene un determinado array, podemos utilizar el atributo `ndim`:

```
import numpy as np

mi_array_1d = np.array([1, 2, 3])
mi_array_2d = np.array([[1, 2, 3], [4, 5, 6]])

print(mi_array_1d.ndim)
print(mi_array_2d.ndim)
```

```
1
2
```

2.2. INDEXADO DE ARRAYS

El indexado de los arrays de NumPy (es decir, el acceso a sus elementos) es similar al indexado de las listas nativas de Python: podemos acceder a un elemento a través del índice de la posición que ocupa dentro del array.

De igual forma que en el resto de colecciones de Python, el índice del primer elemento de un array en NumPy es el cero.

```
import numpy as np

mi_array = np.array([1, 2, 3])

print(mi_array[0])
print(mi_array[1])
print(mi_array[2])
```

```
1
2
3
```

Para indexar arrays de más de una dimensión necesitamos hacer algo similar, pero utilizando dos índices (para entendernos: el primer índice representa la “fila” de la matriz, y el segundo índice representa la “columna”). Por ejemplo, si quiero imprimir por pantalla el primer elemento de la segunda fila, debo hacer lo siguiente:

```
import numpy as np

mi_array = np.array([[1, 2, 3], [4, 5, 6]])

print(mi_array[1, 0])
```

```
1
2
3
```

De forma equivalente, para indexar un elemento de una matriz de tres dimensiones, necesitaremos utilizar tres índices, y así sucesivamente:

```
import numpy as np

mi_array = np.array([[[1, 2, 3],[4, 5, 6]], [1, 2, 3],[4, 5, 6]]])

print(mi_array[1, 0, 2])
```

```
3
```

Al igual que en el Python nativo, los arrays de NumPy también permiten utilizar índices negativos, que empiezan a contar desde el final del array.

Hasta ahora hemos hablado únicamente de acceder a un solo elemento del array cada vez. Sin embargo, también podemos utilizar rangos de índices para acceder a más de un elemento cada vez:

```
import numpy as np

mi_array = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

print(mi_array[3:8])
```

```
[4 5 6 7 8]
```

Los rangos de índices pueden tener 2 o 3 índices. Si tienen 2 índices, el primero representa el inicio y el segundo representa el final:

[inicio:final]

En este tipo de rangos, el primer elemento al que accedemos será el que le corresponde el índice “inicio” y el último elemento será el anterior al índice “final”. Es decir, cogemos desde “inicio” hasta “final”, pero sin el elemento de la posición “final”. También tenemos rangos de 3 índices:

[inicio:final:paso]

Estos rangos son similares a los anteriores, pero aquí iremos cogiendo cada “paso” elementos. Por ejemplo, si yo tengo el rango

[3:8:2], significa que cogeré el elemento de la posición “3”, después el elemento de la posición “5” y, por último, el elemento de la posición “7”.

2.3. LOS MÉTODOS COPY Y VIEW

Aunque ambos métodos de los arrays de NumPy son similares, la realidad es que tienen una diferencia clave: al utilizar `copy()` crearemos un nuevo array idéntico, mientras que al utilizar `view()` obtendremos una vista del array original.

Cuando utilizamos `copy()` para crear una copia y modificamos el array original, no se modificará el nuevo array creado:

```
import numpy as np

mi_array = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
mi_nuevo_array = mi_array.copy()

mi_array[1] = 11

print(mi_array)
print(mi_nuevo_array)
```

```
[1 11 3 4 5 6 7 8 9 10]
[1 2 3 4 5 6 7 8 9 10]
```

Sin embargo, si hacemos lo mismo, pero utilizando `view()`, sí que se verá modificado el nuevo array creado:

```
import numpy as np

mi_array = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
mi_nuevo_array = mi_array.view()

mi_array[1] = 11

print(mi_array)
print(mi_nuevo_array)
```

```
[1 11 3 4 5 6 7 8 9 10]
[1 11 3 4 5 6 7 8 9 10]
```

De forma equivalente, cualquier cambio efectuado sobre el nuevo array creado usando `view()` también se verá reflejado en el array original:

```
import numpy as np

mi_array = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
mi_nuevo_array = mi_array.view()

mi_nuevo_array[1] = 11

print(mi_array)
print(mi_nuevo_array)
```

```
[1 11 3 4 5 6 7 8 9 10]
[1 11 3 4 5 6 7 8 9 10]
```

2.4. EL “SHAPE” DE UN ARRAY

El “shape” o “forma” de un array de NumPy es el número de elementos que tiene en cada dimensión. Todos los arrays de NumPy tienen el atributo “shape”, que nos permite conocer esta información:

```
import numpy as np

mi_array = np.array([[1, 2, 3], [4, 5, 6]])

print(mi_array.shape)
```

```
(2, 3)
```

En este caso, el resultado “(2, 3)” nos indica que el array tiene dos dimensiones, en la que la primera dimensión tiene 2 elementos y la segunda dimensión tiene 3.

NumPy nos permite también modificar el “shape” de un determinado array, añadiendo o dimensiones o modificando el número de elementos en cada dimensión. En el ejemplo inferior cambiamos un array de 1 dimensión con 9 elementos por una matriz de 2 dimensiones con tamaño 3x3:

```
import numpy as np

mi_array = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])

mi_nuevo_array = mi_array.reshape(3, 3)

print(mi_array)
print(mi_nuevo_array)
```

```
[1 2 3 4 5 6 7 8 9]
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Cualquier cambio de “shape” es posible siempre y cuando el número de elementos antes y después del cambio de tamaño coincidan.

2.5. ITERANDO SOBRE UN ARRAY DE NUMPY

Como sabéis, iterar sobre una colección de datos significa recorrerla y pasar por todos sus elementos uno a uno. Aunque existen formas más avanzadas de hacerlo, en la gran mayoría de los casos vamos a iterar sobre un array NumPy utilizando un bucle for (de igual forma que utilizamos un bucle for para recorrer una lista):

```
import numpy as np

mi_array = np.array([1, 2, 3, 4, 5, 6, 7])

for elemento in mi_array:
   print(elemento)
```

```
1
2
3
4
5
6
7
```

Sin embargo, si queremos iterar sobre los elementos de un array de dos dimensiones, tenemos que hacer una cosa distinta ya que, como

vemos en el cuadro inferior, si hacemos lo mismo obtendremos un resultado diferente:

```
import numpy as np

mi_array = np.array([[1, 2, 3], [4, 5, 6]])

for elemento in mi_array:
   print(elemento)
```

```
[1 2 3]
[4 5 6]
```

Para acceder a los elementos, necesitamos dos bucles for anidados, uno que recorra una dimensión (filas) y otro que recorra la otra dimensión (columnas):

```
import numpy as np

mi_array = np.array([[1, 2, 3], [4, 5, 6]])

for fila in mi_array:
   for elemento in fila:
      print(elemento)
```

```
1
2
3
4
5
6
```

De forma equivalente, para acceder a los elementos de una matriz de 3 dimensiones necesitamos utilizar tres bucles for anidados:

```
import numpy as np

mi_array = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10,
11, 12]]])

for matriz in mi_array:
   for fila in matriz:
      for elemento in fila:
         print(elemento)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
```

2.6. UNIENDO ARRAYS DE NUMPY

Al unir arrays, juntamos el contenido de dos o más arrays en un solo array. En NumPy, esta unión se hace en base a uno de los ejes (una de las dimensiones). Por defecto, en caso de no especificar dimensión, se tomará la dimensión (eje) 0.

Para la unión, utilizamos el método `concatenate()`:

```
import numpy as np

mi_array1 = np.array([1, 2, 3])
mi_array2 = np.array([4, 5, 6])

mi_array_res = np.concatenate((mi_array1, mi_array2))

print(mi_array1)
print(mi_array2)
print(mi_array_res)
```

```
[1 2 3]
[4 5 6]
[1 2 3 4 5 6]
```

Probemos ahora a unir arrays de dos dimensiones y así entenderemos cómo funciona el parámetro `axis` y las dimensiones:

```
import numpy as np

mi_array1 = np.array([[1, 2, 3], [4, 5, 6]])
mi_array2 = np.array([[7, 8, 9], [10, 11, 12]])

mi_array_res = np.concatenate((mi_array1, mi_array2), axis=0)

print(mi_array_1)
print(mi_array_2)
print(mi_array_res)
```

```
[[1 2 3]
 [4 5 6]]
[ 7  8  9]
[10 11 12]]
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

```
import numpy as np

mi_array1 = np.array([[1, 2, 3], [4, 5, 6]])
mi_array2 = np.array([[7, 8, 9], [10, 11, 12]])

mi_array_res = np.concatenate((mi_array1, mi_array2), axis=1)

print(mi_array_1)
print(mi_array_2)
print(mi_array_res)
```

```
[[1 2 3]
 [4 5 6]]
[ 7  8  9]
[10 11 12]]
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
```

Al igual que podemos unir arrays, también los podemos separar, utilizando la función `array_split()`, a la que tendremos que pasarle el número de arrays en el que queremos dividir el array original:

```
import numpy as np

mi_array = np.array([1, 2, 3, 4, 5, 6])

mi_nuevo_array = np.array_split(mi_array, 3)

print(mi_array)
print(mi_nuevo_array)
```

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

Como podemos ver en el código superior, el resultado que obtenemos es un array que contiene los diferentes arrays en los que se ha dividido, de forma que luego nosotros podemos acceder a ellos utilizando índices, tal y como hemos aprendido en apartados anteriores.

Si intentamos dividir un array en un número de arrays de forma que sus elementos no se puedan dividir de forma equitativa, no pasa nada, NumPy tomará las decisiones oportunas por nosotros:

```
import numpy as np

mi_array = np.array([1, 2, 3, 4, 5, 6])

mi_nuevo_array = np.array_split(mi_array, 4)

print(mi_array)
print(mi_nuevo_array)
```

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

2.7. BÚSQUEDA DE ELEMENTOS EN UN ARRAY DE NUMPY

NumPy también nos ofrece una función para poder buscar los elementos de un array que cumplen una determinada condición. Esta función es `where()` y devuelve los índices de los elementos que cumplen dicha condición:

```
import numpy as np

mi_array = np.array([1, 2, 3, 4, 3, 2, 1])

indices = np.where(mi_array == 2)

print(indices)
```

```
(array([1, 5]),)
```

El resultado de esta función es una tupla, pero que solo tiene valores en su primer elemento (es una forma un poco rara, pero es la que ha escogido NumPy para hacerlo).

2.8. ORDENANDO UN ARRAY DE NUMPY

Al igual que existe un método para ordenar las listas en Python, también existe otro para ordenar los elementos de un array de NumPy. Este método es `sort()`:

```
import numpy as np

mi_array = np.array([4, 2, 3, 1])

mi_array_ordenado = np.sort(mi_array)

print(mi_array)
print(mi_array_ordenado)
```

```
[4 2 3 1]
[1 2 3 4]
```

Al igual que con las listas, el criterio de ordenación para los arrays con números es siempre de menor a mayor, y con cadenas de caracteres siempre en órden alfabético, aunque esto se puede modificar (pero es algo más avanzado que escapa al carácter básico de este manual). En caso de querer ordenar arrays de más dimensiones, siempre se ordenarán los elementos en base a la última dimensión:

```
import numpy as np

mi_array = np.array([[4, 2, 3], [1, 3, 2]])

mi_array_ordenado = np.sort(mi_array)

print(mi_array)
print(mi_array_ordenado)
```

```
[[2 3 4]
 [1 2 3]]
```

3. EL MÓDULO RANDOM DE NUMPY

La librería NumPy nos ofrece el módulo `random` para poder trabajar con números aleatorios (realmente pseudoaleatorios, porque los ordenadores como tal no pueden generar números aleatorios sin información externa, pero a partir de ahora haremos referencia a ellos como “aleatorios”).

3.1. NÚMEROS ENTEROS ALEATORIOS

La función `randint()` nos permite generar un número entero aleatorio:

```
from numpy import random

numero = random.randint(10)
print(numero)

numero = random.randint(10)
print(numero)

numero = random.randint(10)
print(numero)

numero = random.randint(10)
print(numero)

numero = random.randint(10)
print(numero)
```

```
1
9
0
2
3
```

Esta función únicamente recibe un argumento, en este caso el “10”, que es el número entero máximo que puede llegar a generar. Es decir, en este caso, cada llamada a la función generará un número entero aleatorio entre 0 y 10.

3.2. NÚMEROS DECIMALES ALEATORIOS

La función `rand()`, a diferencia de la anterior, nos permite generar un número `float` aleatorio entre 0 y 1:

```
from numpy import random

numero = random.rand()
print(numero)

numero = random.rand()
print(numero)

numero = random.rand()
print(numero)

numero = random.rand()
print(numero)

numero = random.rand()
print(numero)
```

```
0.5506958626451769
0.9483748374930321
0.9548543232763282
0.1126372839845033
0.5495492324343432
```

3.3. GENERACIÓN DE NÚMEROS ALEATORIOS AL MISMO TIEMPO

Tanto la función `randint()` como la función `rand()` nos permiten especificar cuántos números aleatorios queremos generar (así nos ahorramos tener que llamar varias veces a la función).

En el caso de la función `randint()`, podemos especificar el número en el parámetro `size`:

```
from numpy import random
```

```
numero = random.randint(10, size=(5))

print(numero)
```

```
[8 0 8 9 6]
```

Podemos generar también matrices de números aleatorios de varias dimensiones:

```
from numpy import random

numero = random.randint(10, size=(5, 5))

print(numero)
```

```
[[6 1 6 4 9]
 [2 7 0 8 4]
 [2 2 8 5 7]
 [1 1 5 6 7]
 [0 8 3 3 9]]
```

En la función `rand()`, especificamos el tamaño directamente:

```
from numpy import random

numero = random.rand(5)

print(numero)
```

```
[0.8578794 0.6283647 0.1393036 0.4389132 0.3643001]
```

De igual forma, podemos generar matrices:

```
from numpy import random

numero = random.rand(5)

print(numero)
```

```
[[0.11255645 0.3956824 ]
 [0.33345394 0.24117891]]
```

3.4. GENERAR NÚMERO ALEATORIO DE UN ARRAY

El método `choice()` recibe un array como argumento de entrada y aleatoriamente devuelve uno de los valores que aparecen en él:

```
from numpy import random

numero = random.choice([1, 3, 2, 4])

print(numero)
```

```
3
```

En este caso también podemos especificar el parámetro size para decirle a NumPy cuántas elecciones queremos que nos haga:

```
from numpy import random

numeros = random.choice([1, 3, 2, 4], size=(2, 3))

print(numeros)
```

```
[[2 1 2]
 [1 4 4]]
```

3.5. ALTERANDO LAS PROBABILIDADES

El método `choice()` nos permite enviarle también una distribución, especificando la probabilidad de salida que debe tener cada elemento del array. En el caso concreto del siguiente ejemplo:

- La probabilidad de escoger un 1 es de 0.3 (30%).
- La probabilidad de escoger un 3 es de 0.1 (10%).
- La probabilidad de escoger un 2 es del 0.6 (60%).
- La probabilidad de escoger un 4 es del 0 (0%).

```
from numpy import random

numeros = random.choice([1, 3, 2, 4], p=[0.3, 0.1, 0.6, 0.0],
size=(10))

print(numeros)
```

```
[1 2 3 3 1 2 2 2 2 2]
```

Como podemos ver, en el anterior ejemplo no se cumple exactamente la distribución esperada:

- Han salido dos unos (20% de la muestra) cuando la probabilidad esperada era del 30%.

- Han salido dos treses (20% de la muestra) cuando la probabilidad esperada era del 10%.

- Han salido seis doses (60% de la muestra) y la probabilidad esperada era del 60%.

  En resumen, en lugar del 1 esperado, ha salido un 3 no esperado.

  ¿Significa eso que está mal?

No. Aquí estamos hablando de probabilidades. Un dado tiene un sexto de probabilidades de sacar cada número, pero eso no obliga a que si tiramos 6 dados, obtengamos 6 números diferentes. Pues aquí ocurre exactamente lo mismo.

El módulo random de NumPy tiene alguna otra funcionalidad interesante, pero para su uso y entendimiento se requieren ciertos conocimientos matemáticos (distribuciones chi-cuadrado distribución de Poisson, etc.), así que no ahondaremos más en ellas en este manual.

4. FUNCIONES ADICIONALES DE NUMPY

Por último, me gustaría dedicar las últimas páginas de esta guía a hablar de algunas funciones adicionales que incorpora NumPy y que nos permiten hacer todavía más sencillo el manejo de sus arrays.

En primer lugar, me gustaría hablar de la función `add()`, que nos permite sumar los valores de dos arrays:

```
from numpy import random

mi_array1 = [1, 2, 3]
mi_array2 = [4, 5, 6]

suma = np.add(mi_array1, mi_array2)

print(suma)
```

```
[5 7 9]
```

También podemos restar arrays:

```
from numpy import random

mi_array1 = [1, 2, 3]
mi_array2 = [4, 5, 6]

resta = np.subtract(mi_array1, mi_array2)

print(resta)
```

```
[-3 -3 -3]
```

Y multiplicarlos:

```
from numpy import random

mi_array1 = [1, 2, 3]
mi_array2 = [4, 5, 6]

multiplicacion = np.multiply(mi_array1, mi_array2)

print(multiplicacion)
```

```
[4 10 18]
```

Y dividirlos:

```
from numpy import random

mi_array1 = [1, 2, 3]
mi_array2 = [4, 5, 6]

division = np.divide(mi_array1, mi_array2)

print(division)
```

```
[0.25 0.4  0.5 ]
```

De forma similar, también podemos elevar unos elementos a otros, calcular su módulo, etc. Por ejemplo, también podemos calcular el valor absoluto de un array:

```
from numpy import random

mi_array = [-1, -2, -3, -8, -99]

valor_abs = np.absolute(mi_array)

print(valor_abs)
```

```
[ 1  2  3  8 99]
```

Existen también algunas funciones para redondear los decimales de un número. La primera es `trunc()`, que elimina los decimales y devuelve el entero más cercano a cero:

```
from numpy import random

mi_array = [-1.26, 4.89]

resultado = np.trunc(mi_array)

print(resultado)
```

```
[-1. 4.]
```

La función around() redondea (con los criterios matemáticos) un número al número de decimales indicado:

```
from numpy import random

resultado = np.around(6.287954, 2)

print(resultado)
```

```
6.29
```

NumPy también nos ofrece implementaciones para las funciones suelo (entero inferior) y techo (entero superior):

```
from numpy import random

resultado = np.floor([-1.26, 4.89])

print(resultado)
```

```
[-2. 4.]
```

```
from numpy import random

resultado = np.ceil([-1.26, 4.89])

print(resultado)
```

```
[-1. 5.]
```

Por último, en la siguiente tabla se detallan algunas de las principales funciones adicionales restantes que incorpora NumPy:

Función	Descripción
`log2()`	Calcula el logaritmo en base 2.
`log10()`	Calcula el logaritmo en base 10.
`log()`	Calcula el logaritmo en base e.
`cumsum()`	Calcula la suma acumulativa de los elementos de un array.
`prod()`	Calcula la multiplicación entre los elementos de uno o varios arrays distintos.
`cumprod()`	Calcula el producto acumulativo de los elementos de un array.
`diff()`	Calcula la diferencia discreta entre los elementos de un array.
`lcm()`	Calcula el mínimo común múltiplo entre dos números.
`gcd()`	Calcula el máximo común divisor entre dos números.
`sin()`	Calcula el valor del seno del valor o valores que recibe.
`cos()`	Calcula el valor del coseno del valor o valores que recibe.
`tan()`	Calcula el valor de la tangente del valor o valores que recibe.
`deg2rad()`	Convierte grados en radianes.
`rad2deg()`	Convierte radianes en grados.
`arcsin()`	Calcula el valor del arcoseno del valor o valores que recibe.
`arccos()`	Calcula el valor del arcocoseno del valor o valores que recibe.
`arctan()`	Calcula el valor de la arcotangente del valor o valores que recibe.
`sinh()`	Calcula el valor del seno hiperbólico del valor o valores que recibe.
`cosh()`	Calcula el valor del coseno hiperbólico del valor o valores que recibe.
`tanh()`	Calcula el valor de la tangente hiperbólica del valor o valores que recibe.
`unique()`	Devuelve los elementos que solo aparecen una vez en un array.

Existen algunos métodos más, pero sus usos son muy específicos y no es demasiado interesante conocerlos.