Vectores y matrices

Introducción a matrices y vectores en NumPy

En matemáticas, un vector es un conjunto ordenado de números. Un vector de tamaño n tiene n números, que se pueden representar como una lista o una matriz de n elementos.

Una matriz es un conjunto de vectores. Una matriz de tamaño m x n tiene m filas y n columnas. Cada fila es un vector de tamaño n.

NumPy es una biblioteca de Python para el procesamiento numérico. NumPy proporciona una variedad de funciones y operaciones para trabajar con matrices y vectores.

Creación de matrices y vectores en NumPy

Para crear una matriz o vector en NumPy, podemos usar la función `array()`. Esta función toma una lista o una tupla de números como entrada y crea una matriz o vector a partir de ellos.

Por ejemplo, para crear un vector de tamaño 5, podemos usar el siguiente código:

```
import numpy as np

# Crea un vector de tamaño 5
v = np.array([1, 2, 3, 4, 5])

# Imprime el vector
print(v)
```

Este código imprimirá el siguiente resultado:

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

Para crear una matriz de tamaño 3 x 2, podemos usar el siguiente código:

```
# Crea una matriz de tamaño 3 x 2
m = np.array([[1, 2], [3, 4], [5, 6]])

# Imprime la matriz
print(m)
```

Este código imprimirá el siguiente resultado:

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

```
 [5 6]]
```

Operaciones con matrices y vectores en NumPy

NumPy proporciona una variedad de funciones y operaciones para trabajar con matrices y vectores.

Algunas de las operaciones más comunes son:

- **Suma:** Suma dos matrices o vectores de igual tamaño.
- **Resta:** Resta dos matrices o vectores de igual tamaño.
- **Multiplicación:** Multiplica dos matrices o vectores de igual tamaño.
- **División:** Divide dos matrices o vectores de igual tamaño.
- **Transposición:** Invierte las filas y columnas de una matriz.
- **Redimensionamiento:** Cambia el tamaño de una matriz o vector.

Operaciones más comunes

Suma:

```
import numpy as np

# Crea dos vectores de tamaño 5
v1 = np.array([1, 2, 3, 4, 5])
v2 = np.array([6, 7, 8, 9, 10])

# Suma los vectores
v3 = v1 + v2

# Imprime el vector resultante
print(v3)
```

Este código imprimirá el siguiente resultado:

```
[7 9 11 13 15]
```

Resta:

```
import numpy as np

# Crea dos vectores de tamaño 5
v1 = np.array([1, 2, 3, 4, 5])
v2 = np.array([6, 7, 8, 9, 10])

# Resta los vectores
v3 = v1 - v2
```

```
# Imprime el vector resultante
print(v3)
```

Este código imprimirá el siguiente resultado:

```
[-5 -5 -5 -5 -5]
```

Multiplicación:

```
import numpy as np

# Crea dos matrices de tamaño 3 x 2
m1 = np.array([[1, 2], [3, 4], [5, 6]])
m2 = np.array([[7, 8], [9, 10], [11, 12]])

# Multiplica las matrices
m3 = m1 * m2

# Imprime la matriz resultante
print(m3)
```

Este código imprimirá el siguiente resultado:

```
[[50 84]
 [126 164]
 [202 244]]
```

División:

```
import numpy as np

# Crea dos matrices de tamaño 3 x 2
m1 = np.array([[1, 2], [3, 4], [5, 6]])
m2 = np.array([[7, 8], [9, 10], [11, 12]])

# Divide las matrices
m3 = m1 / m2

# Imprime la matriz resultante
print(m3)
```

Este código imprimirá el siguiente resultado:

```
[[0.14285714 0.25]
 [0.33333333 0.4]
 [0.42857143 0.5]]
```

Transposición:

```
import numpy as np

# Crea una matriz de tamaño 3 x 2
m = np.array([[1, 2], [3, 4], [5, 6]])

# Transpone la matriz
mt = m.transpose()

# Imprime la matriz transpuesta
print(mt)
```

Este código imprimirá el siguiente resultado:

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

Redimensionamiento:

```
import numpy as np

# Crea un vector de tamaño 5
v = np.array([1, 2, 3, 4, 5])

# Redimensiona el vector a tamaño 3 x 2
v = v.reshape(3, 2)

# Imprime el vector redimensionado
print(v)
```

Este código imprimirá el siguiente resultado:

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

Funciones para generar arrays

NumPy proporciona una variedad de funciones para generar arrays. Estas funciones se utilizan para crear arrays de diferentes tipos, tamaños y formas.

Funciones para generar arrays de un solo valor

- `np.ones()`: Crea un array de un solo valor.
- `np.zeros()`: Crea un array de un solo valor.
- `np.full()`: Crea un array de un solo valor.

Funciones para generar arrays de rangos

- `np.arange()`: Crea un array de números en un rango.
- `np.linspace()`: Crea un array de números equiespaciados en un rango.
- `np.logspace()`: Crea un array de números con escala logarítmica en un rango.

Funciones para generar arrays de números aleatorios

- `np.random.rand()`: Crea un array de números aleatorios uniformemente distribuidos.
- `np.random.randn()`: Crea un array de números aleatorios normalizados.
- `np.random.randint()`: Crea un array de números aleatorios enteros.

Funciones para generar arrays de matrices

- `np.eye()`: Crea una matriz identidad.
- `np.diag()`: Crea una matriz diagonal.
- `np.vander()`: Crea una matriz Vandermonde.

Funciones para generar arrays estructurados

- `np.zeros_like()`: Crea un array estructurado con los mismos tipos de datos que un array existente.
- `np.ones_like()`: Crea un array estructurado con los mismos tipos de datos que un array existente.
- `np.full_like()`: Crea un array estructurado con los mismos tipos de datos que un array existente.

Ejemplos

```
import numpy as np

# Crea un array de un solo valor
arr = np.ones(5)
print(arr)

# Crea un array de rango
arr = np.arange(10)
print(arr)

# Crea un array de números aleatorios
arr = np.random.rand(5)
print(arr)

# Crea una matriz identidad
arr = np.eye(3)
print(arr)
```

```
# Crea una matriz diagonal
arr = np.diag([1, 2, 3])
print(arr)

# Crea una matriz Vandermonde
arr = np.vander([1, 2, 3], 3)
print(arr)
```

Este código imprimirá los siguientes resultados:

```
[1. 1. 1. 1. 1.]
[0 1 2 3 4 5 6 7 8 9]
[0.46211716 0.65084663 0.83957509 0.9746692  1.0692377  1.1292377]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[1. 0. 0.]
 [0. 2. 0.]
 [0. 0. 3.]]
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]]
```

Indexación y segmentación

La indexación y la segmentación son dos técnicas de programación que se utilizan para acceder a elementos individuales o subconjuntos de una estructura de datos.

Indexación

La indexación se utiliza para acceder a un elemento individual de una estructura de datos. Se realiza mediante el uso de corchetes `[]` para especificar el índice del elemento deseado.

En el contexto de NumPy, la indexación se utiliza para acceder a un elemento individual de un array. Por ejemplo, el siguiente código accede al primer elemento de un array de 5 elementos:

```
import numpy as np

arr = np.arange(5)

# Accede al primer elemento
print(arr[0])
```

Este código imprimirá el siguiente resultado:

```
0
```

También se puede acceder a elementos de un array utilizando índices negativos. Los índices negativos se cuentan desde el final del array.

Por ejemplo, el siguiente código accede al último elemento de un array de 5 elementos:

```
import numpy as np

arr = np.arange(5)

# Accede al último elemento
print(arr[-1])
```

Este código imprimirá el siguiente resultado:

```
4
```

Segmentación

La segmentación se utiliza para acceder a un subconjunto de una estructura de datos. Se realiza mediante el uso de corchetes `[start:end]`, donde `start` es el índice del primer elemento del subconjunto y `end` es el índice del último elemento del subconjunto.

En el contexto de NumPy, la segmentación se utiliza para acceder a un subconjunto de un array. Por ejemplo, el siguiente código accede a los primeros tres elementos de un array de 5 elementos:

```
import numpy as np

arr = np.arange(5)

# Accede a los primeros tres elementos
print(arr[0:3])
```

Este código imprimirá el siguiente resultado:

```
[0 1 2]
```

También se puede utilizar la segmentación para acceder a un subconjunto de elementos de un array utilizando índices negativos.

Por ejemplo, el siguiente código accede a los últimos tres elementos de un array de 5 elementos:

```
import numpy as np

arr = np.arange(5)

# Accede a los últimos tres elementos
print(arr[-3:])
```

Este código imprimirá el siguiente resultado:

```
[2 3 4]
```

Segmentación con step

La segmentación también se puede utilizar para acceder a un subconjunto de elementos de un array utilizando un paso. Se realiza mediante el uso de corchetes `[start:end:step]`, donde `start` es el índice del primer elemento del subconjunto, `end` es el índice del último elemento del subconjunto y `step` es el tamaño del paso.

Por ejemplo, el siguiente código accede a los elementos de un array de 5 elementos con un paso de 2:

```
import numpy as np

arr = np.arange(5)

# Accede a los elementos con un paso de 2
print(arr[::2])
```

Este código imprimirá el siguiente resultado:

```
[0 2 4]
```

La indexación y la segmentación son dos técnicas de programación esenciales para trabajar con estructuras de datos. En el contexto de NumPy, se utilizan para acceder a elementos individuales o subconjuntos de arrays.

Reorganización

La reorganización y la ampliación son dos técnicas que se utilizan para cambiar el tamaño o la forma de un array de NumPy.

Reorganización

La reorganización se utiliza para cambiar la forma de un array sin cambiar los datos que contiene. Se realiza mediante el uso del método `reshape()`.

El método `reshape()` toma como entrada una tupla que especifica la nueva forma del array. Los elementos de la tupla representan el número de elementos en cada dimensión del nuevo array.

Por ejemplo, el siguiente código crea un array de 5 elementos:

```
import numpy as np

arr = np.arange(5)

print(arr)
```

Este código imprimirá el siguiente resultado:

```
[0 1 2 3 4]
```

El siguiente código reorganiza el array en un array de 2 dimensiones con 2 filas y 2 columnas:

```
arr = arr.reshape(2, 2)

print(arr)
```

Este código imprimirá el siguiente resultado:

```
[[0 1]
 [2 3]]
```

El método `reshape(-1,1)` de NumPy se utiliza para cambiar la forma de un array de vector a matriz. El parámetro `-1` en `reshape(-1, 1)` indica que el primer índice de la nueva forma se calculará automáticamente para que el número total de elementos del array sea el mismo que el número original de elementos. El segundo índice de la nueva forma siempre será 1.

Por ejemplo, el siguiente código crea un array de 5 elementos:

```
import numpy as np

arr = np.arange(5)

print(arr)
```

Este código imprimirá el siguiente resultado:

```
```

```
[0 1 2 3 4]
```

El siguiente código utiliza `reshape(-1, 1)` para convertir el array en un array de una sola dimensión:

```
arr = arr.reshape(-1, 1)

print(arr)
```

Este código imprimirá el siguiente resultado:

```
[[0]
 [1]
 [2]
 [3]
 [4]]
```

En este caso, el primer índice de la nueva forma es 5, que es el número total de elementos del array original. El segundo índice de la nueva forma es 1, que es el número de columnas del array nuevo.

El método `reshape(-1, 1)` se puede utilizar para convertir un array multidimensional en un array unidimensional. También se puede utilizar para convertir un array unidimensional en un array bidimensional con una sola columna.

Operaciones vectorizadas

Las operaciones vectorizadas son operaciones que se aplican a todos los elementos de un array de forma simultánea. Esto se diferencia de las operaciones escalares, que se aplican a un solo elemento de un array a la vez.

Las operaciones vectorizadas son más eficientes que las operaciones escalares porque se pueden realizar en una sola operación en lugar de en una operación por elemento. Esto puede suponer una mejora significativa del rendimiento, especialmente cuando se trabaja con arrays de gran tamaño.

En NumPy, la mayoría de las operaciones matemáticas y lógicas son vectorizadas. Por ejemplo, la siguiente operación suma todos los elementos de un array:

```
import numpy as np

arr = np.arange(5)

print(arr.sum())
```

Este código imprimirá el siguiente resultado:

```
10
```

Esta operación se realiza en una sola operación, en lugar de en una operación por elemento.

Las operaciones vectorizadas también se pueden utilizar para realizar operaciones más complejas, como la multiplicación matricial o la resolución de sistemas de ecuaciones lineales.

Ventajas de las operaciones vectorizadas

Las operaciones vectorizadas ofrecen las siguientes ventajas:

- Son más eficientes que las operaciones escalares.
- Son más fáciles de escribir y leer.
- Son más portátiles.

Desventajas de las operaciones vectorizadas

Las operaciones vectorizadas pueden tener las siguientes desventajas:

- Pueden requerir más memoria.
- Pueden ser menos flexibles que las operaciones escalares.

Operaciones aritméticas vectorizadas

Las operaciones aritméticas vectorizadas son operaciones que se aplican a todos los elementos de un array de forma simultánea. Esto se diferencia de las operaciones escalares, que se aplican a un solo elemento de un array a la vez.

Las operaciones aritméticas vectorizadas son más eficientes que las operaciones escalares porque se pueden realizar en una sola operación en lugar de en una operación por elemento. Esto puede suponer una mejora significativa del rendimiento, especialmente cuando se trabaja con arrays de gran tamaño.

En NumPy, la mayoría de las operaciones matemáticas y lógicas son vectorizadas. Por ejemplo, la siguiente operación suma todos los elementos de un array:

```
import numpy as np

arr = np.arange(5)

print(arr.sum())
```

Este código imprimirá el siguiente resultado:

```
10
```

Esta operación se realiza en una sola operación, en lugar de en una operación por elemento.

Las operaciones aritméticas vectorizadas también se pueden utilizar para realizar operaciones más complejas, como la multiplicación matricial o la resolución de sistemas de ecuaciones lineales.

Ventajas de las operaciones aritméticas vectorizadas

Las operaciones aritméticas vectorizadas ofrecen las siguientes ventajas:

- Son más eficientes que las operaciones escalares.
- Son más fáciles de escribir y leer.
- Son más portátiles.

Desventajas de las operaciones aritméticas vectorizadas

Las operaciones aritméticas vectorizadas pueden tener las siguientes desventajas:

- Pueden requerir más memoria.
- Pueden ser menos flexibles que las operaciones escalares.

Ejemplos

El siguiente código suma dos arrays de igual tamaño:

```
import numpy as np

arr1 = np.arange(5)
arr2 = np.arange(5, 10)

print(arr1 + arr2)
```

Este código imprimirá el siguiente resultado:

```
[5 6 7 8 9]
```

La operación `+` se aplica a todos los elementos de los dos arrays de forma simultánea.

El siguiente código multiplica un array por un escalar:

```
import numpy as np

arr = np.arange(5)

print(arr * 2)
```

Este código imprimirá el siguiente resultado:

```
[0 2 4 6 8]
```

La operación * se aplica a todos los elementos del array por el escalar 2.

Aquí hay otro ejemplo de una operación aritmética vectorizada:

Python

```
import numpy as np

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

print(arr / 2)
```

Este código imprimirá el siguiente resultado:

```
[0.5 1. 1.5 2. 2.5]
```

La operación / se aplica a todos los elementos del array por el escalar 2.

Aquí hay otro ejemplo:

```
import numpy as np

arr1 = np.array([1, 2, 3, 4, 5])
arr2 = np.array([6, 7, 8, 9, 10])

print(arr1 * arr2)
```

Este código imprimirá el siguiente resultado:

```
[6 14 24 36 50]
```

La operación * se aplica a todos los elementos de los dos arrays de forma simultánea.

Aquí hay un ejemplo final:

```
import numpy as np

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

print(arr.max())
```

Este código imprimirá el siguiente resultado:

5

La operación `max()` se aplica a todos los elementos del array y devuelve el valor máximo.

Estos son solo algunos ejemplos de las muchas operaciones aritméticas vectorizadas que se pueden realizar en arrays de NumPy.

Funciones punto a punto

Las funciones punto a punto son funciones que se aplican a todos los elementos de un array de forma simultánea. Esto se diferencia de las funciones escalares, que se aplican a un solo elemento de un array a la vez.

Las funciones punto a punto son más eficientes que las funciones escalares porque se pueden realizar en una sola operación en lugar de en una operación por elemento. Esto puede suponer una mejora significativa del rendimiento, especialmente cuando se trabaja con arrays de gran tamaño.

En NumPy, hay muchas funciones punto a punto disponibles, que incluyen funciones matemáticas, lógicas y estadísticas.

Ejemplos

El siguiente código aplica la función `sin()` a todos los elementos de un array:

```
import numpy as np

arr = np.arange(5)

print(np.sin(arr))
```

Este código imprimirá el siguiente resultado:

```
[0.0 0.84147098 0.90929743 0.14112000 -0.75680249]
```

La función `sin()` se aplica a todos los elementos del array de forma simultánea.

El siguiente código aplica la función `log()` a todos los elementos de un array:

```
import numpy as np

arr = np.arange(5)

print(np.log(arr))
```

Este código imprimirá el siguiente resultado:

```
[0.0 0.69314718 1.09861229 1.38629436 1.60943791]
```

La función `log()` se aplica a todos los elementos del array de forma simultánea.

Estos son solo algunos ejemplos de las muchas funciones punto a punto que se pueden aplicar a arrays de NumPy.

Funciones de agregación

Las funciones de agregación son funciones que operan sobre un conjunto de datos y devuelven un solo valor. Las funciones de agregación se pueden utilizar para calcular estadísticas básicas, como la suma, la media, la mediana y la desviación estándar.

En NumPy, hay muchas funciones de agregación disponibles. Las funciones de agregación más comunes son:

- `sum()`: Devuelve la suma de todos los elementos de un array.
- `mean()`: Devuelve la media de todos los elementos de un array.
- `median()`: Devuelve la mediana de todos los elementos de un array.
- `std()`: Devuelve la desviación estándar de todos los elementos de un array.

Ejemplos

El siguiente código calcula la suma de todos los elementos de un array:

```
import numpy as np

arr = np.arange(5)

print(np.sum(arr))
```

Este código imprimirá el siguiente resultado:

```
10
```

El siguiente código calcula la media de todos los elementos de un array:

```
import numpy as np

arr = np.arange(5)

print(np.mean(arr))
```

Este código imprimirá el siguiente resultado:

```
2.5
```

El siguiente código calcula la mediana de todos los elementos de un array:

```
import numpy as np

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

print(np.median(arr))
```

Este código imprimirá el siguiente resultado:

```
3
```

El siguiente código calcula la desviación estándar de todos los elementos de un array:

```
import numpy as np

arr = np.arange(5)

print(np.std(arr))
```

Este código imprimirá el siguiente resultado:

```
1.5811388300841896
```

Estos son solo algunos ejemplos de las muchas funciones de agregación que se pueden aplicar a arrays de NumPy.

Ventajas de las funciones de agregación

Las funciones de agregación ofrecen las siguientes ventajas:

- Son fáciles de usar.
- Son rápidas y eficientes.
- Son versátiles.

Desventajas de las funciones de agregación

Las funciones de agregación pueden tener las siguientes desventajas:

- Pueden no ser adecuadas para todos los conjuntos de datos.
- Pueden ser menos flexibles que las funciones punto a punto.

Expresiones condicionales

Los arrays booleanos en NumPy son arrays que contienen valores booleanos, que pueden ser `True` o `False`. Los arrays booleanos se pueden utilizar para realizar operaciones condicionales, como la selección de elementos de un array que cumplen una condición determinada.

Las expresiones condicionales en NumPy se utilizan para evaluar una condición y devolver un valor booleano. Las expresiones condicionales se pueden utilizar para controlar el flujo de un programa o para realizar operaciones condicionales en arrays.

Arrays booleanos

Los arrays booleanos se pueden crear utilizando la función `np.full()` o la función `np.zeros()`. La función `np.full()` crea un array con todos los elementos establecidos en un valor booleano determinado. La función `np.zeros()` crea un array con todos los elementos establecidos en `False`.

Ejemplos

El siguiente código crea un array booleano con todos los elementos establecidos en `True`:

```
import numpy as np

arr = np.full(5, True)

print(arr)
```

Este código imprimirá el siguiente resultado:

```
[ True True True True True]
```

El siguiente código crea un array booleano con todos los elementos establecidos en `False`:

```
import numpy as np

arr = np.zeros(5, dtype=bool)

print(arr)
```

Este código imprimirá el siguiente resultado:

```
[False False False False False]
```

Operaciones condicionales

Las operaciones condicionales se pueden utilizar para evaluar una condición y devolver un valor booleano. Las operaciones condicionales más comunes son:

- `>`: Devuelve `True` si el primer operando es mayor que el segundo operando.
- `<`: Devuelve `True` si el primer operando es menor que el segundo operando.
- `==`: Devuelve `True` si los dos operandos son iguales.
- `!=`: Devuelve `True` si los dos operandos son diferentes.

Ejemplos

El siguiente código utiliza la operación `>` para evaluar si un número es mayor que 10:

```
import numpy as np

num = 12

print(num > 10)
```

Este código imprimirá el siguiente resultado:

```
True
```

El siguiente código utiliza la operación `<=` para evaluar si un número es menor o igual que 5:

```
import numpy as np

num = 5

print(num <= 5)
```

Este código imprimirá el siguiente resultado:

```
True
```

Expresiones condicionales en arrays

Las expresiones condicionales se pueden utilizar para realizar operaciones condicionales en arrays. El siguiente código utiliza una expresión condicional para seleccionar los elementos de un array que son mayores que 5:

```
import numpy as np

arr = np.arange(10)
```

```
print(arr[arr > 5])
```

Este código imprimirá el siguiente resultado:

```
[6 7 8 9]
```

El siguiente código utiliza una expresión condicional para calcular la media de los elementos de un array que son mayores que 5:

```
import numpy as np

arr = np.arange(10)

print(np.mean(arr[arr > 5]))
```

Este código imprimirá el siguiente resultado:

```
7.5
```

Los arrays booleanos y las expresiones condicionales son herramientas poderosas que se pueden utilizar para realizar operaciones complejas en arrays de NumPy.

Operaciones en conjuntos

La tabla que proporcionaste muestra las funciones de NumPy para operar conjuntos. Estas funciones se pueden utilizar para realizar una variedad de operaciones en conjuntos de datos, como encontrar los elementos únicos de un conjunto, probar la existencia de un elemento en un conjunto, encontrar la intersección o la diferencia de dos conjuntos, y unir dos conjuntos.

Aquí hay algunos ejemplos de cómo usar estas funciones:

```
import numpy as np

# Crear dos arrays
arr1 = np.array([1, 2, 3, 4, 5])
arr2 = np.array([2, 4, 5, 6, 7])

# Encontrar los elementos únicos de un conjunto
unique_elements = np.unique(arr1)

# Probar la existencia de un elemento en un conjunto
element = 3
exists_in_array = np.in1d(element, arr1)
```

```
# Encontrar la intersección de dos conjuntos
intersection = np.intersect1d(arr1, arr2)

# Encontrar la diferencia de dos conjuntos
difference = np.setdiff1d(arr1, arr2)

# Unir dos conjuntos
union = np.union1d(arr1, arr2)

# Imprimir los resultados
print("Elementos únicos de arr1:", unique_elements)
print("El elemento {} existe en arr1: {}".format(element, exists_in_array))
print("Intersección de arr1 y arr2:", intersection)
print("Diferencia de arr1 y arr2:", difference)
print("Unión de arr1 y arr2:", union)
```

Salida:

```
Elementos únicos de arr1: [1 2 3 4 5]
El elemento 3 existe en arr1: True
Intersección de arr1 y arr2: [2 4 5]
Diferencia de arr1 y arr2: [1 3]
Unión de arr1 y arr2: [1 2 3 4 5 6 7]
```

Las funciones de NumPy para operar conjuntos son una herramienta poderosa que se puede utilizar para manipular conjuntos de datos de forma eficiente y eficaz.

Operaciones comunes con arrays

np.transpose

La función `np.transpose()` invierte los ejes de un array. Esto significa que las filas se convierten en columnas y las columnas se convierten en filas.

Ejemplo:

```
import numpy as np

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

print(arr)
print(np.transpose(arr))
```

Salida:

```

```

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

np.fliplr/np.flipud

Las funciones `np.fliplr()` y `np.flipud()` invierten los elementos de un array en cada fila o columna, respectivamente.

Ejemplo:

```
import numpy as np

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

print(arr)
print(np.fliplr(arr))
print(np.flipud(arr))
```

Salida:

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

np.rot90

La función `np.rot90()` rota los elementos de un array 90 grados a lo largo de los dos primeros ejes.

Ejemplo:

```
import numpy as np

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

print(arr)
print(np.rot90(arr))
```

Salida:

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

np.sort

La función `np.sort()` ordena los elementos de un array a lo largo de un eje especificado. De forma predeterminada, el eje es el último eje del array.

Ejemplo:

```
import numpy as np

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

print(arr)
print(np.sort(arr))
```

Salida:

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

También podemos especificar el eje a lo largo del cual queremos ordenar el array:

```
import numpy as np

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

print(arr)
print(np.sort(arr, axis=1))
```

Salida:

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

Operaciones matriciales

Hasta ahora hemos discutido los arrays N-dimensionales generales. Una de las principales aplicaciones de estos arrays es representar los conceptos matemáticos de vectores, matrices y tensores, y en este caso de uso, también necesitamos frecuentemente calcular operaciones vectoriales y matriciales como productos escalares (internos), productos de puntos (matriciales) y productos tensoriales (externos).

np.dot

La función `np.dot()` calcula el producto matricial de dos arrays. El producto matricial es una operación que se realiza entre dos matrices para producir una nueva matriz.

Ejemplo:

```
import numpy as np

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

print(np.dot(A, B))
```

Salida:

```
[[19 22]
 [43 50]]
```

También podemos utilizar `np.dot()` para calcular el producto escalar de dos vectores. El producto escalar es una operación que se realiza entre dos vectores para producir un número escalar.

Ejemplo:

```
import numpy as np

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

print(np.dot(a, b))
```

Salida:

```
32
```

np.inner

La función `np.inner()` calcula el producto interno de dos arrays. El producto interno es una operación que se realiza entre dos vectores para producir un número escalar.

Ejemplo:

```
import numpy as np

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

print(np.inner(a, b))
```

Salida:

```
32
```

np.cross

La función `np.cross()` calcula el producto vectorial de dos arrays. El producto vectorial es una operación que se realiza entre dos vectores para producir un nuevo vector.

Ejemplo:

```
import numpy as np

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

print(np.cross(a, b))
```

Salida:

```
[-3  6 -3]
```

np.tensordot

La función `np.tensordot()` calcula el producto tensorial de dos arrays. El producto tensorial es una operación que se realiza entre dos arrays para producir un nuevo array.

Ejemplo:

```
import numpy as np

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

print(np.tensordot(a, b, axes=(1, 0)))
```

Salida:

```
[[19 22]
 [43 50]]
```

np.outer

La función `np.outer()` calcula el producto externo de dos arrays. El producto externo es una operación que se realiza entre dos arrays para producir un nuevo array.

Ejemplo:

```
import numpy as np

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

print(np.outer(a, b))
```

Salida:

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

np.kron

La función `np.kron()` calcula el producto de Kronecker de dos arrays. El producto de Kronecker es una operación que se realiza entre dos arrays para producir un nuevo array.

Ejemplo:

```
import numpy as np

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

print(np.kron(a, b))
```

Salida:

```
[[5 6 10 12]
 [15 18 30 36]
```

```
 [25 30 50 60]
 [35 42 70 84]]
```

np.einsum

La función `np.einsum()` calcula la convención de sumación de Einstein para arrays multidimensionales. La convención de sumación de Einstein es una forma de especificar cómo se deben sumar los índices de un array multidimensional.

Ejemplo:

```
import numpy as np

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

print(np.einsum('ij,j->i', a, b))
```

Salida:

```
[19 22]
```

Estas son solo algunas de las operaciones que se pueden realizar en vectores y matrices con NumPy. NumPy también proporciona una amplia gama de otras funciones para trabajar con arrays,

Resumen

En esta unidad, hemos visto los siguientes temas relacionados con NumPy:

- **Arrays N-dimensionales:** NumPy proporciona una estructura de datos llamada array para representar datos numéricos multidimensionales. Los arrays N-dimensionales se pueden crear usando la función `np.array()`.
- **Operaciones en arrays:** NumPy proporciona una amplia gama de funciones para realizar operaciones en arrays, como suma, resta, multiplicación, división, etc.
- **Operaciones en conjuntos:** NumPy proporciona funciones para realizar operaciones en conjuntos de datos, como encontrar los elementos únicos de un conjunto, probar la existencia de un elemento en un conjunto, encontrar la intersección o la diferencia de dos conjuntos, y unir dos conjuntos.
- **Operaciones en vectores y matrices:** NumPy proporciona funciones para realizar operaciones en vectores y matrices, como el producto escalar, el producto matricial, el producto vectorial, el producto tensorial, etc.

En resumen, NumPy es una biblioteca de Python que proporciona una amplia gama de funciones para trabajar con datos numéricos. NumPy es una herramienta poderosa que se puede utilizar para una variedad de aplicaciones, como análisis de datos, aprendizaje automático e informática científica.

Aquí hay algunos ejemplos de cómo se pueden utilizar las funciones de NumPy que hemos visto hasta este punto:

- **Arrays N-dimensionales:** Podemos utilizar arrays N-dimensionales para representar datos numéricos de la vida real, como imágenes, audio y video.
- **Operaciones en arrays:** Podemos utilizar operaciones en arrays para realizar cálculos matemáticos en datos numéricos.
- **Operaciones en conjuntos:** Podemos utilizar operaciones en conjuntos para manipular datos categóricos.
- **Operaciones en vectores y matrices:** Podemos utilizar operaciones en vectores y matrices para realizar cálculos geométricos y algebraicos.

En los próximos temas, veremos más funciones de NumPy que nos permitirán realizar operaciones más complejas en datos numéricos.