Numpy

Operaciones con arrays

CertiDevs

Índice de contenidos

1. Introducción

Numpy ofrece una amplia gama de operaciones y funciones que puedes realizar en **arrays**.

- **Creación de arrays**: Crear arrays de diferentes dimensiones, formas y tipos de datos.
- **Indexación y selección**: Acceder a elementos específicos, filas, columnas o secciones de un array.
- **Slicing**: Extraer subarrays o segmentos de un array.
- **Manipulación de la forma del array**: Cambiar la forma, redimensionar o transponer arrays.
- **Operaciones aritméticas**: Suma, resta, multiplicación, división y otras operaciones aritméticas entre arrays o entre un array y un escalar.
- **Operaciones de álgebra lineal**: Producto escalar, producto vectorial, multiplicación de matrices, inversa de matrices, cálculo de autovalores y autovectores, y descomposición de matrices.
- **Operaciones lógicas**: Aplicar operaciones lógicas como AND, OR, NOT y XOR a arrays.
- **Operaciones de comparación**: Comparar arrays elemento por elemento para obtener arrays de booleanos.
- **Funciones matemáticas**: Aplicar funciones matemáticas como seno, coseno, exponencial, logaritmo y raíz cuadrada a los elementos de un array.
- **Estadísticas y agregaciones**: Calcular la suma, el promedio, la mediana, la varianza, la desviación estándar, el mínimo, el máximo y otros estadísticos sobre un array.
- **Ordenamiento y búsqueda**: Ordenar un array, buscar el k-ésimo elemento más pequeño o encontrar los índices de los elementos que cumplen con cierta condición.
- **Broadcasting**: Realizar operaciones entre arrays de diferentes dimensiones y formas de manera eficiente.
- **Operaciones de conjunto**: Encontrar la unión, la intersección y la diferencia entre dos arrays.
- **Concatenación y división**: Unir o dividir arrays a lo largo de un eje específico.
- **Funciones universales (ufunc)**: Aplicar funciones vectorizadas a arrays para un rendimiento y eficiencia optimizados.
- **Masking**: Filtrar y seleccionar elementos de un array utilizando máscaras booleanas.
- **Fancy indexing**: Acceder a elementos no contiguos de un array utilizando listas o arrays de índices.

2. Creación de arrays

2.1. Array a partir de una lista

Puedes crear un array de Numpy a partir de una lista de Python usando la función `np.array()`:

```
lst = [1, 2, 3, 4, 5]
```

```
arr = np.array(lst)
print(arr)
```

2.2. Array de ceros

Para crear un **array de ceros**, puedes usar la función `np.zeros()` y proporcionar la forma del array como argumento:

```
zeros_arr = np.zeros((3, 4))
print(zeros_arr)
```

2.3. Array de unos

De manera similar, puedes crear un array de unos usando la función np.ones():

```
ones_arr = np.ones((2, 3))
print(ones_arr)
```

2.4. Array vacío

La función `np.empty()` crea un array sin inicializar sus elementos a ningún valor en particular.

Es más rápido que crear un array de ceros o unos, pero los valores iniciales serán arbitrarios:

```
empty_arr = np.empty((2, 2))
print(empty_arr)
```

2.5. Array con valores espaciados uniformemente

La función `np.arange()` crea un array con valores espaciados uniformemente entre un rango dado:

```
arange_arr = np.arange(0, 10, 2)  # Empieza en 0, termina antes de 10, con un paso de
2
print(arange_arr)
```

2.6. Array con un número específico de puntos

La función `np.linspace()` crea un array con un número específico de puntos espaciados uniformemente entre un rango dado:

```
linspace_arr = np.linspace(0, 1, 5)  # Empieza en 0, termina en 1, con 5 puntos en
```

```
total
print(linspace_arr)
```

2.7. Array con valores aleatorios

Numpy también proporciona funciones para crear arrays con valores aleatorios.

Por ejemplo, puedes crear un array con valores aleatorios uniformemente distribuidos entre 0 y 1 usando la función `np.random.rand()`:

```
random_arr = np.random.rand(2, 3)
print(random_arr)
```

2.8. Array con valores aleatorios siguiendo una distribución normal

La función `np.random.randn()` crea un array con **valores aleatorios** siguiendo una **distribución normal** (media 0, desviación estándar 1):

```
normal_arr = np.random.randn(4, 4)
print(normal_arr)
```

2.9. Array con valores enteros aleatorios

La función `np.random.randint()` crea un array con valores enteros aleatorios dentro de un rango especificado:

```
int_arr = np.random.randint(0, 10, size=(3, 3))  # Valores aleatorios
```

3. Indexación y selección

La **indexación** y **selección** en NumPy te permite acceder a elementos específicos, filas, columnas o secciones de un array. A continuación, se presentan algunos ejemplos de código para diferentes casos:

```
import numpy as np

# Crear un array 1D
arr_1d = np.array([1, 2, 3, 4, 5])

# Indexación en un array 1D
print(arr_1d[0])  # Primer elemento
```

```
print(arr_1d[-1])  # Último elemento

# Crear un array 2D (matriz)
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Indexación en un array 2D
print(arr_2d[0, 0])  # Primer elemento de la primera fila
print(arr_2d[1, 2])  # Tercer elemento de la segunda fila
print(arr_2d[-1, -1])  # Último elemento de la última fila

# Selección de filas y columnas en un array 2D
print(arr_2d[0, :])  # Primera fila
print(arr_2d[:, 1])  # Segunda columna

# Selección de secciones (slicing) en un array 1D
print(arr_1d[1:4])  # Elementos del segundo al cuarto (índices 1, 2 y 3)

# Selección de secciones (slicing) en un array 2D
print(arr_2d[0:2, 0:2])  # Submatriz 2x2 en la esquina superior izquierda
```

4. Slicing

El **slicing** es una técnica para **extraer subarrays** o segmentos de un array.

A continuación, se presentan algunos ejemplos de código para diferentes casos de slicing en NumPy:

```
import numpy as np

# Crear un array 1D
arr_1d = np.array([1, 2, 3, 4, 5])

# Slicing en un array 1D
print(arr_1d[1:4])  # Elementos del segundo al cuarto (índices 1, 2 y 3)
print(arr_1d[:3])  # Elementos desde el principio hasta el tercer elemento (índices 0,
1 y 2)
print(arr_1d[2:])  # Elementos desde el tercer elemento hasta el final (índices 2, 3 y
4)

# Crear un array 2D (matriz)
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Slicing en un array 2D (selección de filas y columnas)
print(arr_2d[:2, :])  # Primeras dos filas
print(arr_2d[:, :2])  # Primeras dos columnas
```

```
# Slicing en un array 2D (submatrices)
print(arr_2d[0:2, 0:2])  # Submatriz 2x2 en la esquina superior izquierda
print(arr_2d[1:, 1:])  # Submatriz 2x2 en la esquina inferior derecha

# Slicing con pasos en un array 1D
print(arr_1d[::2])  # Elementos en posiciones pares (índices 0, 2 y 4)
print(arr_1d[::-1])  # Elementos en orden inverso

# Slicing con pasos en un array 2D
print(arr_2d[::2, ::2])  # Elementos en posiciones pares de filas y columnas
(submatriz 2x2 con elementos en las esquinas)
```

5. Manipulación de arrays

NumPy ofrece varias funciones para **cambiar la forma**, **redimensionar** y **aplanar** arrays.

A continuación, se presentan ejemplos de código para diferentes casos de manipulación de la forma del array:

```
import numpy as np

# Crear un array 1D
arr_1d = np.array([1, 2, 3, 4, 5, 6])

# Crear un array 2D (matriz) a partir de un array 1D
arr_2d = arr_1d.reshape(2, 3)
print(arr_2d)
# Output:
# [[1 2 3]
#  [4 5 6]]

# Transponer un array 2D (intercambiar filas y columnas)
arr_transposed = arr_2d.T
print(arr_transposed)
# Output:
# [[1 4]
#  [2 5]
#  [3 6]]

# Aplanar un array 2D en un array 1D
arr_flattened = arr_2d.flatten()
print(arr_flattened)
# Output:
# [1 2 3 4 5 6]

# Cambiar la forma de un array utilizando np.newaxis para agregar una dimensión
arr_expanded = arr_1d[:, np.newaxis]
print(arr_expanded)
```

```
# Output:
# [[1]
#  [2]
#  [3]
#  [4]
#  [5]
#  [6]]

# Redimensionar un array (cambia la forma, pero no garantiza la conservación de los
datos)
arr_resized = np.resize(arr_1d, (2, 4))
print(arr_resized)
# Output:
# [[1 2 3 4]
#  [5 6 1 2]]

# Concatenar dos arrays 1D en un array 2D
arr_1 = np.array([1, 2, 3])
arr_2 = np.array([4, 5, 6])
arr_concatenated = np.stack((arr_1, arr_2))
print(arr_concatenated)
# Output:
# [[1 2 3]
#  [4 5 6]]
```

Reshape y aplanar: Cambiar la forma de un array de Numpy y aplanar un array multidimensional en un array unidimensional.

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

# Cambiar la forma de la matriz a un array 1D (aplanar)
matriz_aplanada = matriz.flatten()
print("Matriz aplanada:", matriz_aplanada)

# Cambiar la forma de la matriz a un array 1D con otra forma
matriz_reshape = matriz.reshape(-1)
print("Matriz con reshape:", matriz_reshape)
```

6. Operaciones aritméticas

Numpy permite realizar **operaciones aritméticas** elemento a elemento en arrays, como **suma**, **resta**, **multiplicación** y **división**.

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

# Suma
```

```
c = a + b
print("Suma:", c)

# Resta
d = a - b
print("Resta:", d)

# Multiplicación
e = a * b
print("Multiplicación:", e)

# División
f = a / b
print("División:", f)
```

Suma acumulada:

```
import numpy as np

# Ventas diarias en la tienda en un período de 7 días (en dólares)
ventas_diarias = np.array([120, 150, 100, 180, 140, 210, 190])

# Calcular la suma acumulativa de las ventas diarias
ventas_acumuladas = np.cumsum(ventas_diarias)

print("Ventas diarias: ", ventas_diarias)
print("Ventas acumuladas: ", ventas_acumuladas)

# Ventas diarias:  [120 150 100 180 140 210 190]
# Ventas acumuladas:  [120 270 370 550 690 900 1090]
```

7. Operaciones matemáticas y trigonométricas

Numpy proporciona **funciones matemáticas** y **trigonométricas** para trabajar con arrays, como *exponentes*, *logaritmos*, *raíces cuadradas* y *funciones trigonométricas*.

```
g = np.array([1, 4, 9])

# Raíz cuadrada
h = np.sqrt(g)
print("Raíz cuadrada:", h)

# Exponente
i = np.exp(g)
print("Exponente:", i)
```

```
# Logaritmo
j = np.log(g)
print("Logaritmo:", j)

# Seno
k = np.sin(g)
print("Seno:", k)

# Coseno
l = np.cos(g)
print("Coseno:", l)
```

8. Operaciones de álgebra lineal

El **álgebra lineal** es una rama de las matemáticas que se ocupa del estudio de los **espacios vectoriales** y las **operaciones lineales** entre ellos, como la suma y el producto escalar.

También aborda conceptos como matrices, sistemas de ecuaciones lineales, determinantes, vectores y valores propios, entre otros.

El álgebra lineal es fundamental en diversas áreas de la ciencia, la ingeniería y la economía, ya que permite modelar y resolver problemas que involucran relaciones lineales entre variables.

Numpy proporciona funciones para realizar operaciones de **álgebra lineal** en arrays, como *producto escalar*, *producto cruzado*, *inversa de matrices* y cálculo de *determinantes*.

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

# Producto escalar
o = np.dot(m, n)
print("Producto escalar:\n", o)

# Inversa de una matriz
p = np.linalg.inv(m)
print("Inversa de una matriz:\n", p)

# Determinante de una matriz
q = np.linalg.det(m)
print("Determinante de una matriz:", q)

# Autovalores y autovectores
r, s = np.linalg.eig(m)
print("Autovalores:", r)
print("Autovectores:\n", s)
```

9. Ordenamiento

NumPy proporciona varias funciones para ordenar elementos, buscar valores específicos y encontrar índices de elementos que cumplen ciertos criterios.

A continuación, se presentan ejemplos de código para diferentes casos de ordenamiento en arrays de NumPy:

Ordenar datos: Ordenar los elementos de un array de Numpy.

```
edades = np.array([25, 30, 18, 40, 35, 22, 45, 50])

# Ordenar edades de menor a mayor
edades_ordenadas = np.sort(edades)
print("Edades ordenadas:", edades_ordenadas)
```

```
import numpy as np

# Crear un array 1D desordenado
arr_1d = np.array([5, 2, 9, 1, 7, 3])

# Ordenar un array 1D (retorna una copia ordenada, el array original no se modifica)
arr_sorted = np.sort(arr_1d)
print(arr_sorted)
# Output:
# [1 2 3 5 7 9]

# Ordenar un array 1D en su lugar (modifica el array original)
arr_1d.sort()
print(arr_1d)
# Output:
# [1 2 3 5 7 9]

# Crear un array 2D desordenado
arr_2d = np.array([[9, 4, 2],
                   [6, 1, 7],
                   [5, 3, 8]])

# Ordenar un array 2D por columnas
arr_sorted_columns = np.sort(arr_2d, axis=0)
print(arr_sorted_columns)
# Output:
# [[5 1 2]
#  [6 3 7]
#  [9 4 8]]

# Ordenar un array 2D por filas
arr_sorted_rows = np.sort(arr_2d, axis=1)
```

```
print(arr_sorted_rows)
# Output:
# [[2 4 9]
#  [1 6 7]
#  [3 5 8]]
```

10. Búsquedas

```
# Buscar el índice del valor mínimo en un array 1D
min_index = np.argmin(arr_1d)
print(min_index)
# Output:
# 0

# Buscar el índice del valor máximo en un array 1D
max_index = np.argmax(arr_1d)
print(max_index)
# Output:
# 5

# Buscar valores que cumplan ciertos criterios en un array
arr_filtered = arr_1d[arr_1d > 3]
print(arr_filtered)
# Output:
# [5 7 9]

# Encontrar los índices de los elementos que cumplen ciertos criterios en un array
indices = np.where(arr_1d > 3)
print(indices)
# Output:
# (array([2, 4, 5]),)
```

Filtrar datos: Seleccionar elementos en un array de Numpy basado en una condición.

```
precios = np.array([50, 100, 150, 200, 250, 300, 350, 400])

# Filtrar precios menores o iguales a 200
precios_filtrados = precios[precios <= 200]
print("Precios filtrados:", precios_filtrados)
```

Reemplazar valores: Reemplazar valores en un array de Numpy basado en una condición.

```
calificaciones = np.array([85, 90, 78, 92, 76, 65, 98, 89])

# Reemplazar calificaciones menores a 80 por 80
calificaciones[calificaciones < 80] = 80
```

```
print("Calificaciones ajustadas:", calificaciones)
```

11. Funciones universales (ufunc)

Las **funciones universales** (ufunc) son funciones que operan sobre arrays de NumPy **elemento por elemento**, de manera similar a las funciones matemáticas de Python.

Las **funciones universales** de NumPy incluyen operaciones matemáticas simples, como suma, resta, multiplicación y división, y funciones trigonométricas, exponenciales y logarítmicas, entre otras.

Las funciones universales de NumPy son **mucho más rápidas** que las funciones de Python equivalentes, ya que están implementadas en C.

Estas funciones son altamente eficientes y se implementan en lenguajes de bajo nivel como C o Fortran para mejorar su rendimiento.

Las ufuncs son útiles para realizar operaciones matemáticas y lógicas en arrays sin necesidad de utilizar **bucles explícitos**.

A continuación, se presentan ejemplos de código que demuestran cómo usar algunas funciones universales en NumPy:

```
import numpy as np

# Crear dos arrays de ejemplo
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([5, 6, 7, 8])

# Operaciones aritméticas básicas (elemento a elemento)
sum_result = np.add(arr1, arr2)
print("Suma:", sum_result)
# Output: Suma: [ 6  8 10 12]

diff_result = np.subtract(arr1, arr2)
print("Resta:", diff_result)
# Output: Resta: [-4 -4 -4 -4]

product_result = np.multiply(arr1, arr2)
print("Producto:", product_result)
# Output: Producto: [ 5 12 21 32]

quotient_result = np.divide(arr1, arr2)
print("Cociente:", quotient_result)
# Output: Cociente: [0.2        0.33333333 0.42857143 0.5       ]

# Funciones trigonométricas
angles = np.array([0, np.pi / 6, np.pi / 4, np.pi / 3])
sin_values = np.sin(angles)
```

```
print("Seno:", sin_values)
# Output: Seno: [0.         0.5        0.70710678 0.8660254 ]

# Funciones exponenciales y logarítmicas
arr3 = np.array([1, 2, 3, 4])
exp_result = np.exp(arr3)
print("Exponencial:", exp_result)
# Output: Exponencial: [ 2.71828183  7.3890561  20.08553692 54.59815003]

log_result = np.log(arr3)
print("Logaritmo natural:", log_result)
# Output: Logaritmo natural: [0.         0.69314718 1.09861229 1.38629436]

# Funciones de redondeo
arr4 = np.array([1.2, 2.5, 3.7, 4.1])
round_result = np.round(arr4)
print("Redondeo:", round_result)
# Output: Redondeo: [1. 2. 4. 4.]

floor_result = np.floor(arr4)
print("Piso:", floor_result)
# Output: Piso: [1. 2. 3. 4.]

ceil_result = np.ceil(arr4)
print("Techo:", ceil_result)
# Output: Techo: [2. 3. 4. 5.]
```

11.1. Diferencia sin usar ufuncs

Para aclarar la diferencia de usar ufuncs a no usarlas, veamos cómo se realizarían las operaciones aritméticas sin usar las ufuncs de NumPy y luego comparemos con las ufuncs:

Sin usar ufuncs:

```
import numpy as np

# Crear dos arrays de ejemplo
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([5, 6, 7, 8])

# Suma (elemento a elemento) sin usar ufuncs
sum_result = np.zeros(len(arr1))
for i in range(len(arr1)):
    sum_result[i] = arr1[i] + arr2[i]
print("Suma sin usar ufuncs:", sum_result)
# Output: Suma sin usar ufuncs: [ 6.  8. 10. 12.]
```

Usando ufuncs:

```
# Suma (elemento a elemento) usando ufuncs
sum_result = np.add(arr1, arr2)
print("Suma usando ufuncs:", sum_result)
# Output: Suma usando ufuncs: [ 6  8 10 12]
```

El resultado es el mismo en ambos casos, pero al usar **ufuncs** se obtiene un código más limpio y eficiente.

Las **ufuncs** están implementadas en **lenguajes de bajo nivel (C, Fortran)** y están optimizadas para mejorar el rendimiento. Además, las ufuncs ofrecen una forma más **concisa** y **legible** de realizar operaciones matemáticas y lógicas en arrays de NumPy.

Siempre que sea posible, se recomienda utilizar **ufuncs** en lugar de **bucles explícitos** en Python.

Cuando usas los operadores aritméticos como `+` y `-` en arrays de NumPy, las operaciones se realizan elemento a elemento, y NumPy llama a las **ufuncs** correspondientes (`np.add()` y `np.subtract()`, respectivamente) de manera implícita.

Por lo tanto, aunque no estés llamando explícitamente a las **ufuncs** en el siguiente código, NumPy las utiliza internamente para realizar las operaciones.

```
# Suma
c = a + b
print("Suma:", c)

# Resta
d = a - b
print("Resta:", d)
```

12. Leer datos de CSV

NumPy proporciona la función `genfromtxt()` para leer datos de archivos CSV.

Aunque Numpy no es la herramienta más adecuada para leer archivos CSV (pues Pandas es más versátil en este sentido), aún así puedes hacerlo usando la función `numpy.genfromtxt()`.

Aquí te muestro un ejemplo de cómo leer un archivo CSV en Numpy:

12.1. Leer archivo CSV en Numpy

```
# Asumiendo que el archivo 'data.csv' contiene datos numéricos separados por comas
data = np.genfromtxt('data.csv', delimiter=',')
```

Algunos parámetros útiles de numpy.genfromtxt son:

- **delimiter**: Especifica el delimitador que separa los valores en el archivo CSV. En este caso,

usamos comas (',').

- **skip_header**: Permite omitir un número específico de líneas al comienzo del archivo (por ejemplo, si hay encabezados en el archivo CSV).
- **dtype**: Puedes especificar el tipo de datos de los elementos del arreglo (por ejemplo, float, int, str), o dejar que Numpy los infiera automáticamente usando None.

Aquí tienes un ejemplo que incluye estos parámetros:

```
data = np.genfromtxt('data.csv', delimiter=',', skip_header=1, dtype=float)
```

Ten en cuenta que, si el archivo CSV contiene datos no numéricos o estructuras más complejas, Numpy podría no ser la mejor opción para leerlo.

12.2. Leer columnas específicas de un archivo CSV

Para leer cada columna de un archivo CSV en arrays distintos de Numpy, primero lee el archivo CSV completo utilizando `numpy.genfromtxt()` y luego separa cada columna en arrays individuales usando la indexación de Numpy.

Supongamos que tienes un archivo CSV con tres columnas numéricas y quieres leer cada columna en un array separado de Numpy.

```
data = np.genfromtxt('data.csv', delimiter=',', skip_header=1, dtype=float)
```

Separa cada columna en un array diferente utilizando la indexación de Numpy:

```
column_1 = data[:, 0]
column_2 = data[:, 1]
column_3 = data[:, 2]
```

Aquí, `data[:, 0]` selecciona todos los elementos de la primera columna (índice `0`), `data[:, 1]` selecciona todos los elementos de la segunda columna (índice `1`), y así sucesivamente.