NumPy

Документація

Сайт

Завантаження

```
pip install numpy
```

Початок роботи

Основним об'єктом NumPy є однорідний багатовимірний масив (у numpy називається numpy.ndarray). Це багатовимірний масив елементів (зазвичай чисел), одного типу.

Найбільш важливі атрибути об'єктів ndarray:

ndarray.ndim - число вимірів (найчастіше їх називають "осі") масиву.

ndarray.shape – розміри масиву, його форма. Це кортеж натуральних чисел, що показує довжину масиву кожної осі. Для матриці з n рядків та m стовпів, shape буде (n,m). Число елементів кортежу shape дорівнює ndim.

ndarray.size – кількість елементів масиву. Очевидно, дорівнює добутку всіх елементів атрибуту shape.

ndarray.dtype – об'єкт, що описує тип елементів масиву. Можна визначити dtype за допомогою стандартних типів даних Python. NumPy тут надає цілий букет можливостей, як вбудованих, наприклад: bool_, character, int8, int16, int32, int64, float8, float16, float32, float64, complex64, object_, і можливість визначити власні типи даних, зокрема і складові.

ndarray.itemsize – розмір кожного елемента масиву в байтах.

ndarray.data – буфер, що містить фактичні елементи масиву. Зазвичай не потрібно використовувати цей атрибут, тому що звертатися до елементів масиву найпростіше за допомогою індексів.

Вимірність масивів

Одним з основних компонентів NumPy є багатовимірні масиви або ndarray (N-dimensional array), які надають потужність та ефективність для роботи з числовими даними.

На відміну від списків тут присутня більше правил та обмежень, які гарантують для нас з вами чіткість роботи та можливість передбачити результат опрацювання даного масиву. Також, за рахунок наявності спеціальних типів даних ми можемо гарантувати роботу з пам'яттю набагато більш ефективними методами ніж у аналогічних структурах.

У масивів NumPy можна працювати з різними вимірами, включаючи одновимірні (вектори), двовимірні (матриці) та вищі виміри. Виміри масивів визначаються їх формою (shape) та розмірністю (dimensionality).

Форма (shape) масиву вказує на кількість елементів у кожному розмірі масиву. Наприклад, масив з формою (3, 4) має 3 рядки та 4 стовпці. Форма масиву може бути отримана за допомогою властивості `shape`.

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 1
2]])
print(arr.shape)  # Вивід: (3, 4)
```

Розмірність (dimensionality) масиву вказує на кількість вимірів або осей, які він має. Наприклад, двовимірний масив має розмірність 2, трирозмірний масив має розмірність 3 і т.д. Розмірність масиву можна отримати за допомогою властивості `ndim`.

```
import numpy as np

arr = np.array([1, 2, 3, 4])
print(arr.ndim)  # Вивід: 1

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(arr.ndim)  # Вивід: 2

arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(arr.ndim)  # Вивід: 3
```

Крім того, NumPy надає можливості для перетворення форми та розмірності масивів за допомогою функцій `reshape()` і `resize()`.

Ці виміри масивів у NumPy дозволяють зручно та ефективно працювати з числовими даними у векторах, матрицях та більш високорозмірних структурах даних.

На практиці частіше за все використовуються різновиди 0, 1, 2 та 3 вимірний масиви. Будь-який інший масив за замовченням називається N вимірним.

0 вимірний масив означає просте число

```
import numpy as np

arr = np.array(42)

print(arr)
```

Одновимірний масив це аналог списку

```
import numpy as np

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

print(arr)
```

Двовимірний масив це список списків, що частіше за все представляє собою відображення будь-якої таблиці. А також в загально математичне практиці відображає матрицю.

```
import numpy as np

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

print(arr)
```

3 та більше вимірні масиви відображають за своєю характеристикою більш складні структури які можуть являти собою топології, таблиці з взаємопов'язаними елементами та багато іншого.

```
import numpy as np

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

print(arr)
```

Зверніть увагу, що всі масиви є симетричними. Це означає що кількість елементів в незалежності від його вимірності має бути однакова для кожної площини. Якщо така дія не буде виконано, ми отримаємо помилку

Створення масиву

У NumPy є багато способів створити масив. Один з найпростіших - створити масив зі звичайних списків або кортежів Python, використовуючи функцію numpy.array() (запам'ятайте: array - функція, що створює об'єкт типу ndarray):

```
import numpy as np
a = np.array([1, 2, 3]) # array([1, 2, 3])
type(a) #<class 'numpy.ndarray'>
```

Так само, функція array() трансформує вкладені послідовності багатомірні масиви даних (List, Set, Tuple). Тип елементів масиву залежить від типу елементів вихідної послідовності (але можна і перевизначити його на момент створення).

```
b = np.array([[1.5, 2, 3], [4, 5, 6]])
"""
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
"""
```

Функція array() не єдина функція створення масивів. Зазвичай елементи масиву спочатку невідомі, а масив, у якому зберігатимуться, вже потрібний. Тому є кілька функцій для того, щоб створювати масиви з якимось вихідним вмістом (за замовчуванням тип масива, що створюється — float64).

Функція zeros() створює масив із нулів

```
np.zeros((3, 5))
```

```
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

Функція ones() генерує масив заданого розміру заповнений одиницями.

```
np.ones((2, 2, 2))
```

```
array([[[ 1.,  1.],
        [ 1.,  1.]],

       [[ 1.,  1.],
        [ 1.,  1.]]])
```

Функція eye() створює одиничну матрицю (двовимірний масив)

```
np.eye(5)
```

```
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

Функція empty() створює масив без заповнення. Вихідний вміст випадковий і залежить від стану пам'яті на момент створення масиву (тобто від сміття, що в ній зберігається):

```
np.empty((3, 3))
```

```
array([[  6.93920488e-310,   6.93920488e-310,   6.93920149e
-310],
```

```
       [  6.93920058e-310,   6.93920058e-310,   6.93920058e
-310],
       [  6.93920359e-310,   0.00000000e+000,   6.93920501e
-310]])
```

Для створення послідовностей чисел, NumPy є функція arange(), аналогічна вбудованій в Python range(), тільки замість списків вона повертає масиви, і приймає не тільки цілі значення:

```
np.arange(10, 30, 5)
```

```
array([10, 15, 20, 25])
```

```
np.arange(0, 1, 0.1)
```

```
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.
8,  0.9])
```

Взагалі, при використанні arange() з аргументами типу float, складно бути впевненим у тому, скільки елементів буде отримано (через обмеження точності чисел із плаваючою комою). Тому, в таких випадках зазвичай краще використовувати функцію linspace(), яка замість кроку як один з аргументів приймає число, що дорівнює кількості потрібних елементів:

```
np.linspace(0, 2, 9)  # 9 чисел від 0 до 2 включно
```

```
array([  0. ,  0.25,  0.5 ,  0.75, 1. , 1.25, 1.5 , 1.75,
2. ])
```

Вивід масивів

```
print(np.arange(0, 3000, 1))

"""[   0    1    2 ..., 2997 2998 2999]"""
```

Якщо розмір масива завеликий (>55 елементів) то система автоматично буде ховати середню частину та виводити лише початкові та кінцеві значення

Індекси масивів

Принципи роботи аналогічні до принципів роботи індекса у рядках чи списках. Однак форма звернення трішечки відрізняється для кращого сприйняття

Для одновимірного:

```
import numpy as np

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

print(arr[0])
```

Для багатовимірного

```
import numpy as np

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

print(arr[0, 1, 2])
# 6
```

0 → [[1, 2, 3], [4, 5, 6]]

1 → [4, 5, 6]

2 → 6

Зрізи масивів

Аналогічно до списків

Має 2 форми: [*start*:*end*] та [*start*:*end*:*step*]

Для одновимірного масиву

```
import numpy as np

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

print(arr[1:5])
```

для багатовимірного масивом

```
import numpy as np

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

print(arr[1, 1:4])
```

Типи даних numpy

У бібліотеці NumPy доступні різноманітні типи даних, які можна використовувати при створенні масивів. Ось декілька поширених типів даних, що підтримуються в NumPy:

1. **bool**: Булевий тип даних (`True` або `False`).
2. **int**: Цілі числа, які можуть бути знаковими або беззнаковими. Наприклад, `int8`, `uint16`, `int32` і т.д., відповідно до розміру в байтах.
3. **float**: Дробові числа з плаваючою комою, такі як `float16`, `float32`, `float64` і т.д., відповідно до розміру в байтах.
4. **complex**: Комплексні числа, які складаються з двох частин: дійсної і уявної.
5. **string**: Рядки тексту.
6. **object**: Об'єктний тип даних, який дозволяє зберігати будь-який Python-об'єкт у масиві NumPy.
7. **datetime**: Дати та часи.

Ці типи даних можуть використовуватись при створенні масивів за допомогою функції `numpy.array()`. Також, NumPy надає можливість явно вказати тип даних при створенні масиву за допомогою аргументу `dtype`.

Наприклад, створення масиву з типом `int32`:

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype=np.int32)
print(arr.dtype)  # Вивід: int32
```

Зауважте, що тип даних в NumPy може впливати на ефективність виконання обчислень та споживання пам'яті. Вибір правильного типу даних залежить від конкретних потреб вашої програми та обсягу даних, з якими ви працюєте.

Також типи даних можна передавати через спеціальні літерами

- `i` - integer
- `b` - boolean
- `u` - unsigned integer
- `f` - float
- `c` - complex float
- `m` - timedelta
- `M` - datetime
- `O` - object
- `S` - string
- `U` - unicode string
- `V` - fixed chunk of memory for other type (void)

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='S')

print(arr)
print(arr.dtype)
```

В NumPy ви можете конвертувати типи даних масиву за допомогою функції `astype()`. Ця функція створює новий масив з таким самим вмістом, але з новим типом даних. Ось приклад конвертації типу даних у масиві:

```
import numpy as np

arr = np.array([1, 2, 3, 4])
print(arr.dtype)  # Вивід: int64

arr = arr.astype(np.float64)
print(arr.dtype)  # Вивід: float64
```

У цьому прикладі спочатку створюється масив з типом `int64`, а потім за допомогою `astype()` конвертується в масив з типом `float64`.

Також важливо зазначити, що при конвертації типу даних може відбутися втрата точності. Наприклад, при конвертації цілих чисел у числа з плаваючою комою може втрачатися десяткова частина.

```
import numpy as np

arr = np.array([1, 2, 3, 4])
print(arr)  # Вивід: [1 2 3 4]

arr = arr.astype(np.float64)
print(arr)  # Вивід: [1. 2. 3. 4.]
```

У цьому прикладі цілі числа конвертуються в числа з плаваючою комою, а десяткова частина відсутня.

Зверніть увагу, що конвертація типу даних створює новий масив, а не змінює початковий масив.

Також зверніть увагу, що при неправильні конвертації, яка не може бути підтримана, наприклад, з рядку у число ('a'→ int) буде помилка

Форма масивів

Форма (shape) масиву визначає його розмірність та розмір у кожному розмірі. Вона представляється у вигляді кортежу цілих чисел і може бути отримана за допомогою атрибуту `shape`.

Ось приклад:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.shape)  # Вивід: (2, 3)
```

У цьому прикладі масив `arr` має форму `(2, 3)`, що означає, що він має 2 рядки та 3 стовпці.

Ви також можете змінювати форму масиву за допомогою методу `reshape()`, який створює новий масив з бажаною формою. Наприклад:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])
reshaped_arr = arr.reshape((2, 3))
print(reshaped_arr)
# Вивід:
# [[1 2 3]
#  [4 5 6]]
```

У цьому прикладі початковий одновимірний масив `arr` перетворюється за допомогою `reshape()` на двовимірний масив `reshaped_arr` з формою `(2, 3)`.

Якщо розмірність та вимір нас не буде співпадати-ви отримаєте помилку

Давайте подивимось більш детально

NumPy надає функцію `reshape()` для зміни форми (розмірності) масиву. Функція `reshape()` створює новий масив з тим самими даними, але з іншою формою. Ось кілька прикладів використання функції `reshape()`:

```
import numpy as np

# Одновимірний масив
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
reshaped_arr = arr.reshape((5, 2))  # Змінюємо форму на (5,
```

```
2)
print(reshaped_arr)
# Вивід:
# [[ 1  2]
#  [ 3  4]
#  [ 5  6]
#  [ 7  8]
#  [ 9 10]]

# Двовимірний масив
arr = np.array([[1, 2, 3], [4, 5, 6]])
reshaped_arr = arr.reshape((3, 2))  # Змінюємо форму на (3,
2)
print(reshaped_arr)
# Вивід:
# [[1 2]
#  [3 4]
#  [5 6]]

# Трирозмірний масив
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
reshaped_arr = arr.reshape((2, 4))  # Змінюємо форму на (2,
4)
print(reshaped_arr)
# Вивід:
# [[1 2 3 4]
#  [5 6 7 8]]
```

У цих прикладах функція `reshape()` змінює форму масиву на нову форму, вказану у вигляді кортежу цілих чисел. Важливо, щоб нова форма була сумісною зі змінюваним масивом, тобто кількість елементів масиву має залишатися незмінною після зміни форми.

Також можна використовувати значення `-1` в кортежі форми для автоматичного визначення розміру відповідного розміру. Наприклад:

```
import numpy as np

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

```
reshaped_arr = arr.reshape((3, -1))  # Змінюємо форму на
(3, -1)
print(reshaped_arr)
# Вивід:
# [[1 2]
#  [3 4]
#  [5 6]]
```

У цьому прикладі `reshape()` автоматично визначає розмір другого розміру, щоб підігнати кількість елементів.

Ітерація масивів

У бібліотеці NumPy є кілька способів ітерації (перебору) елементів масиву. Ось кілька популярних методів:

1. **Цикл for**: Можна використовувати цикл `for` для ітерації по кожному елементу масиву:

```
import numpy as np

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

for element in arr:
    print(element)
# Вивід:
# 1
# 2
# 3
# 4
# 5
```

2. **Функція `nditer()`**: Функція `nditer()` дозволяє ітерувати по елементам масиву NumPy з використанням багатьох різних режимів ітерації:

```
import numpy as np

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

```
for element in np.nditer(arr):
    print(element)
# Вивід:
# 1
# 2
# 3
# 4
```

3. **Цикл по рядках (axis=0) або стовпцях (axis=1)**: За допомогою параметра `axis` можна ітерувати по рядках або стовпцях масиву:

```
import numpy as np

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

# Ітерація по рядках
for row in arr:
    print(row)
# Вивід:
# [1 2]
# [3 4]
# [5 6]

# Ітерація по стовпцях
for column in arr.T:
    print(column)
# Вивід:
# [1 3 5]
# [2 4 6]
```

Ці методи дозволяють ефективно перебирати елементи масиву NumPy. Залежно від конкретних потреб, ви можете вибрати найбільш підходящий метод для вашого випадку ітерації.

Об'єднання масивів

У NumPy є кілька функцій для об'єднання (злиття) масивів. Ось кілька популярних методів для об'єднання масивів:

`np.concatenate()`: Функція `np.concatenate()` дозволяє об'єднувати масиви вздовж заданої осі. Вона приймає кортеж масивів, які потрібно об'єднати, і вказується в якому напрямку (осі) потрібно здійснити об'єднання. Ось приклад:

```
import numpy as np

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

result = np.concatenate((arr1, arr2))
print(result)
# Вивід: [1 2 3 4 5 6]
```

`np.vstack()` і `np.hstack()`: Функції `np.vstack()` (vertical stack) та `np.hstack()` (horizontal stack) використовуються для об'єднання масивів вертикально (по рядках) і горизонтально (по стовпцях) відповідно. Ось приклади:

```
import numpy as np

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

# Об'єднання вертикально
result = np.vstack((arr1, arr2))
print(result)
# Вивід:
# [[1 2 3]
#  [4 5 6]]

# Об'єднання горизонтально
result = np.hstack((arr1, arr2))
print(result)
# Вивід: [1 2 3 4 5 6]
```

`np.column_stack()`: Функція `np.column_stack()` використовується для об'єднання масивів як стовпців нового масиву. Ось приклад:

```
import numpy as np

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

result = np.column_stack((arr1, arr2))
print(result)
# Вивід:
# [[1 4]
#  [2 5]
#  [3 6]]
```

Ці методи дозволяють ефективно об'єднувати масиви NumPy в різних комбінаціях. Вибір конкретного методу залежить від потреб вашої задачі та потрібного типу об'єднання.

Розділення масивів

У бібліотеці NumPy є кілька функцій для розбиття (розділення) масиву на менші частини. Ось кілька популярних методів розбиття масивів:

`np.split()`: Функція `np.split()` розділяє масив на підмасиви вздовж заданої осі за допомогою списку індексів розбиття. Ось приклад:

```
import numpy as np

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

result = np.split(arr, [2, 4])  # Розділяємо масив після 2-го та 4-го елементів
print(result)
# Вивід: [array([1, 2]), array([3, 4]), array([5, 6])]
```

`np.array_split()`: Функція `np.array_split()` розділяє масив на підмасиви вздовж заданої осі, при цьому дозволяє різну кількість елементів у

підмасивах. Ось приклад:

```
import numpy as np

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

result = np.array_split(arr, 3)  # Розділяємо масив на 3 пі
дмасиви
print(result)
# Вивід: [array([1, 2, 3]), array([4, 5, 6]), array([7])]
```

`np.hsplit()` і `np.vsplit()`: Функції `np.hsplit()` (horizontal split) і `np.vsplit()` (vertical split) використовуються для розділення масиву горизонтально (по стовпцях) і вертикально (по рядках) відповідно. Ось приклади:

```
import numpy as np

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

# Розділення горизонтально
result = np.hsplit(arr, 3)
print(result)
# Вивід: [array([[1],
#                [4]]),
#         array([[2],
#                [5]]),
#         array([[3],
#                [6]])]

# Розділення вертикально
result = np.vsplit(arr, 2)
print(result)
# Вивід: [array([[1, 2, 3]]),
#         array([[4, 5, 6]])]
```

Ці методи дозволяють ефективно розбивати масиви NumPy на менші частини залежно від потреб вашої задачі.

Сортування масивів

У бібліотеці NumPy є кілька функцій для сортування масивів. Ось декілька популярних методів сортування:

`np.sort()`: Функція `np.sort()` сортує масив в порядку зростання елементів по кожному рядку або по заданій осі. Оригінальний масив залишається незмінним. Ось приклад:

```
import numpy as np

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

# Сортування в порядку зростання
sorted_arr = np.sort(arr)
print(sorted_arr)
# Вивід: [1 2 3 4 5]
```

`np.argsort()`: Функція `np.argsort()` повертає індекси, які б відповідали сортуванню масиву в порядку зростання. Ось приклад:

```
import numpy as np

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

# Сортування індексів
indices = np.argsort(arr)
print(indices)
# Вивід: [1 2 0 4 3]
```

`np.sort()` **з параметром** `axis`: Якщо масив має більше одного розміру, можна вказати параметр `axis`, щоб визначити ось сортування. Ось приклад:

```
import numpy as np

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

```
# Сортування кожного рядка масиву
sorted_arr = np.sort(arr, axis=1)
print(sorted_arr)
# Вивід: [[1 2 3]
#         [4 5 6]]
```

`np.sort()` **з параметром** `kind`: Параметр `kind` визначає алгоритм сортування. За замовчуванням використовується швидке сортування (`'quicksort'`). Інші можливі значення включають `'heapsort'` і `'mergesort'`. Ось приклад:

```
import numpy as np

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

# Сортування з використанням алгоритму сортування злиттям
(mergesort)
sorted_arr = np.sort(arr, kind='mergesort')
print(sorted_arr)
# Вивід: [1 2 3 4 5]
```

Ці методи дозволяють сортувати масиви NumPy залежно

від потреб вашої задачі. Вибір конкретного методу залежить від типу сортування, який ви хочете виконати.

Пошук у массивах

В бібліотеці NumPy є кілька функцій для пошуку елементів у масиві. Ось декілька популярних методів пошуку:

`np.where()`: Функція `np.where()` дозволяє знайти індекси елементів у масиві, які задовольняють певне умову. Ось приклад:

```
import numpy as np

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

```
# Пошук елементів більше за 3
indices = np.where(arr > 3)
print(indices)
# Вивід: (array([3, 4, 5]),)
```

`np.argmax()` і `np.argmin()`: Функції `np.argmax()` та `np.argmin()` дозволяють знайти індекс максимального і мінімального елементів у масиві відповідно. Ось приклад:

```
import numpy as np

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

# Пошук індексу максимального елемента
max_index = np.argmax(arr)
print(max_index)
# Вивід: 4

# Пошук індексу мінімального елемента
min_index = np.argmin(arr)
print(min_index)
# Вивід: 0
```

`np.nonzero()`: Функція `np.nonzero()` повертає індекси ненульових елементів у масиві. Ось приклад:

```
import numpy as np

arr = np.array([0, 2, 0, 4, 0, 6])

# Пошук індексів ненульових елементів
indices = np.nonzero(arr)
print(indices)
# Вивід: (array([1, 3, 5]),)
```

Метод `searchsorted()` у бібліотеці NumPy використовується для знаходження індексу, на якому потрібний елемент масиву повинен бути вставлений, щоб зберегти впорядкованість масиву.

Синтаксис методу `searchsorted()` виглядає так:

```
numpy.searchsorted(a, v, side='left', sorter=None)
```

де:

- `a` - вхідний впорядкований масив, в якому шукаємо;
- `v` - значення, яке потрібно знайти і вставити;
- `side` (необов'язковий) - специфікує, з якого боку вставити значення, якщо він вже присутній у масиві. Можливі значення: `'left'` (зліва) або `'right'` (справа). За замовчуванням встановлено `'left'`.
- `sorter` (необов'язковий) - масив індексів, який вказує порядок елементів `a`. Це дає змогу ефективніше використовувати `searchsorted()` для багаторазового пошуку в тому ж впорядкованому масиві.

Метод `searchsorted()` повертає індекс, на якому потрібний елемент `v` повинен бути вставлений у масив `a`, щоб зберегти впорядкованість. Залежно від значення `side`, цей індекс може бути першим (зліва) або останнім (справа) індексом, де елемент `v` може бути знайдений у масиві `a`. Якщо `v` вже присутній у масиві `a`, метод `searchsorted()` поверне індекс першого (зліва) або останнього (справа) входження цього елемента.

Ось приклад використання методу `searchsorted()`:

```
import numpy as np

arr = np.array([1, 3, 5, 7, 9])

# Знаходження індексу для вставки елемента 6, щоб зберегти
впорядкованість
index = np.searchsorted(arr, 6)
print(index)
# Вивід: 3

# Знаходження індексу для вставки елемента 4 зліва
index = np.searchsorted(arr, 4, side='left')
print(index)
# Вивід:
```

```
2

# Знаходження індексу для вставки елемента 4 справа
index = np.searchsorted(arr, 4, side='right')
print(index)
# Вивід: 2
```

У цьому прикладі метод `searchsorted()` знаходить індекс, на якому елементи можуть бути вставлені у впорядкований масив `arr`.

Важливо! Такий метод має бути використана відсортованому значенні!

Ці методи дозволяють ефективно шукати елементи у масиві NumPy залежно від потреб вашої задачі. Вибір к онкретного методу залежить від типу пошуку, який ви хочете виконати.

Фільтрування масивів

У бібліотеці NumPy можна фільтрувати масиви за допомогою логічних умов. Це дозволяє вибрати лише ті елементи масиву, які відповідають певним критеріям. Ось кілька способів фільтрації масиву NumPy:

1. **Використання логічних умов**: Можна використовувати логічні умови для створення булевого масиву, який вказує, які елементи задовольняють певний критерій. Ось приклад:

```
import numpy as np

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

# Фільтрація елементів більше за 3
filtered_arr = arr[arr > 3]
print(filtered_arr)
# Вивід: [4 5]
```

1. **Використання функції `np.where()`**: Функція `np.where()` може бути використана для фільтрації елементів на основі певної умови. Ось приклад:

```
import numpy as np
```

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

# Фільтрація елементів більше за 3 з використанням np.where
()
filtered_arr = np.where(arr > 3, arr, 0)
print(filtered_arr)
# Вивід: [0 0 0 4 5]
```

У цьому прикладі, якщо умова `arr > 3` є істинною, елементи зберігаються в результатовому масиві, інакше встановлюється значення 0.

Ці методи дозволяють фільтрувати масиви NumPy залежно від потреб вашої задачі і вибрати лише ті елементи, які вам потрібні.

Broadcasting

Broadcasting в NumPy є потужним механізмом, який дозволяє виконувати операції між масивами різних розмірів, але з певними умовами на сумісність.

Умова broadcasting в NumPy:

1. Якщо розмірність двох масивів однакова вздовж всіх осей, то вони є сумісними.
2. Якщо розмірність одного з масивів дорівнює 1 вздовж певної осі, то він може бути розтягнутий (broadcasted) для відповідності розміру іншого масиву вздовж цієї осі.
3. Якщо після розтягнення розмір двох масивів вздовж будь-якої осі співпадає, то вони є сумісними і можна виконувати операцію.

Broadcasting дозволяє виконувати операції на масивах з різними розмірностями без явного повторення чи розширення масивів. Це забезпечує більш компактний та ефективний спосіб програмування.

Розглянемо декілька прикладів:

```
import numpy as np

# Приклад 1: Broadcasting між масивом розміром (3, 3) і мас
ивом розміром (3, 1)
a = np.array([[1, 2, 3],
```

```
              [4, 5, 6],
              [7, 8, 9]])

b = np.array([[10],
              [20],
              [30]])

c = a + b
print(c)
# Вивід:
# [[11 12 13]
#  [24 25 26]
#  [37 38 39]]

# Приклад 2: Broadcasting між масивом розміром (2, 2) і ска
лярним значенням
d = np.array([[1, 2],
              [3, 4]])

e = 10

f = d + e
print(f)
# Вивід:
# [[11 12]
#  [13 14]]

# Приклад 3: Broadcasting між масивами різних розмірів (2,
3) і (3,)
g = np.array([[1, 2, 3],
              [4, 5, 6]])

h = np.array([10, 20, 30])

i = g * h
print

(i)
```

```
# Вивід:
# [[10 40 90]
#  [40 100 180]]
```

В прикладах вище, ми бачимо, що NumPy автоматично розширює масиви для відповідності розмірів і виконує операції між ними. Broadcasting може використовуватись з різними операціями, такими як додавання, віднімання, множення, ділення та іншими.

Це допомагає спростити і зробити код більш ефективним, оскільки не потрібно розширювати або повторювати масиви вручну. NumPy самостійно вирішує, як розширити масиви для виконання операцій.

Це лише загальна інформація про broadcasting у NumPy. Є багато більше можливостей і правил, які дозволяють використовувати broadcasting для вирішення різних завдань з обробки даних.

Термін "broadcasting" описує те, як NumPy обробляє масиви з різними формами під час арифметичних операцій. З дотриманням певних обмежень, менший масив "розтягується" ("broadcasts") по більшому масиву, щоб вони мали сумісні форми. Broadcasting надає можливість векторизувати операції з масивами, щоб циклічні обчислення відбувалися в C, а не в Python. Він робить це без зайвого копіювання даних і зазвичай приводить до ефективних реалізацій алгоритмів. Проте, існують випадки, коли broadcasting є поганою ідеєю, оскільки він призводить до неефективного використання пам'яті, що уповільнює обчислення.

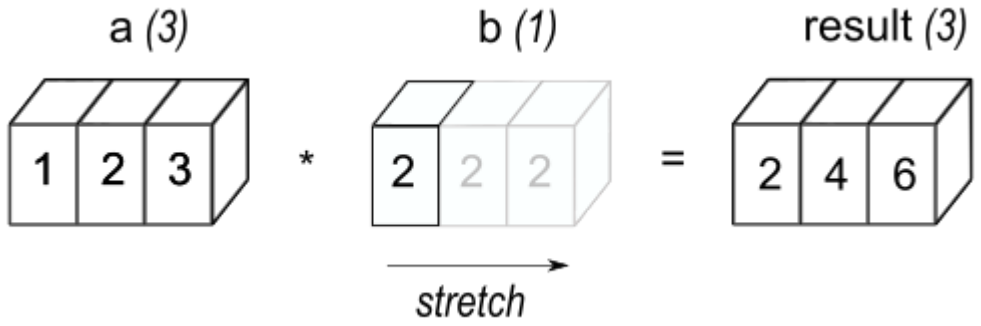
Зазвичай операції у NumPy виконуються над парами масивів на елементарному рівні. В найпростішому випадку два масиви повинні мати точно однакову форму, як у наступному прикладі:

```
a = np.array([1.0, 2.0, 3.0])
b = np.array([2.0, 2.0, 2.0])
a * b
array([2., 4., 6.])
```

Правило broadcasting в NumPy послаблює це обмеження, коли форми масивів відповідають певним умовам. Найпростіший приклад broadcasting виникає, коли масив і скалярне значення комбінуються в операції:

```
a = np.array([1.0, 2.0, 3.0])
b = 2.0
a * b
array([2., 4., 6.])
```

Результат еквівалентний попередньому прикладу, де b було масивом. Ми можемо уявляти, що скалярне значення b розтягується під час арифметичної операції до масиву з такою ж формою, як у масиву a. Нові елементи в b, як показано на малюнку 1, є просто копіями початкового скаляру. Аналогія з розтягуванням є концептуальною. NumPy достатньо розумний, щоб використовувати початкове скалярне значення без фактичного копіювання, щоб операції broadcasting були як ефективними за споживанням пам'яті та обчислювальними ресурсами можливими.



У найпростішому прикладі broadcasting, скаляр b розтягується, щоб стати масивом з такою ж формою, як у масиву a, щоб форми були сумісні для множення елементів один до одного.

Код у другому прикладі є ефективнішим, ніж у першому, оскільки broadcasting переміщує менше пам'яті під час множення (b є скаляром, а не масивом).

Загальні правила broadcasting:

При обробці двох масивів NumPy порівнює їх форми на рівні елементів. Порівняння починається з останньої (найправішої) вимірності і працює відправляючись вліво. Дві вимірності є сумісними, коли:

вони рівні, або
одна з них дорівнює 1.

Якщо ці умови не виконуються, виникає виняток ValueError: operands could not be broadcast together, що вказує на несумісні форми масивів.

Вхідні масиви не обов'язково мають однакову кількість вимірів. Результатуючий масив матиме таку ж кількість вимірів, як у вхідного масиву з найбільшою кількістю вимірів, де розмір кожного виміру є найбільшим серед відповідних вимірів вхідних масивів. Зауважте, що відсутні виміри вважаються розміром одиниці.

Наприклад, якщо у вас є масив RGB значень розміром 256x256×3, і ви хочете масштабувати кожний колір на зворотній коефіцієнт, ви можете помножити зображення на одновимірний масив із 3 значеннями. Порівнявши розміри останніх вимірів цих масивів за правилами broadcasting, ми бачимо, що вони є сумісними:

Зображення (3D масив): 256 x 256 × 3

Масштаб (1D масив): 3
Результат (3D масив): 256 x 256 × 3
Коли одна з порівнюваних вимірностей дорівнює одиниці, використовується інша вимірність. Іншими словами, вимірності з розміром 1 розтягуються або "копіюються", щоб відповідати іншим вимірностям.

У наступному прикладі обидва масиви A і B мають вимірності з довжиною одиниці, які розширяються до більшої форми під час broadcast операції:

A (4D масив): 8 × 1 × 6 × 1
B (3D масив): 7 × 1 × 5
Результат (4D масив): 8 x 7 × 6 × 5
Масиви, які можуть бути broadcast до однакової форми, називаються "broadcastable".

Наприклад, якщо a.shape дорівнює (5,1), b.shape дорівнює (1,6), c.shape дорівнює (6,), а d.shape дорівнює () (тобто d є скаляром), то a, b, c і d можуть бути broadcast до форми (5,6), і:

a діє як (5,6) масив, де a[:,0] розтягується на інші стовпці,
b діє як (5,6) масив, де b[0,:] розтягується на інші рядки,
c діє як (1,6) масив, а тому як (5,6) масив, де c[:] розтягується на кожен рядок, і нарешті,
d діє як (5,6) масив, де одне значення повторюється.

Ось ще декілька прикладів:

A (2D масив): 5 × 4
B (1D масив): 1
Результат (2D масив): 5 x 4

A (2D масив): 5 × 4
B (1D масив): 4
Результат (2D масив): 5 x 4

A (3D масив): 15 × 3 × 5
B (3D масив): 15 × 1 × 5
Результат (3D масив):

15 × 3 × 5

A (3D масив): 15 × 3 × 5
B (2D масив): 3 × 5
Результат (3D масив): 15 x 3 × 5

A (3D масив): 15 × 3 × 5
B (2D масив): 3 × 1
Результат (3D масив): 15 x 3 × 5
Ось приклади форм, які не підлягають broadcasting:

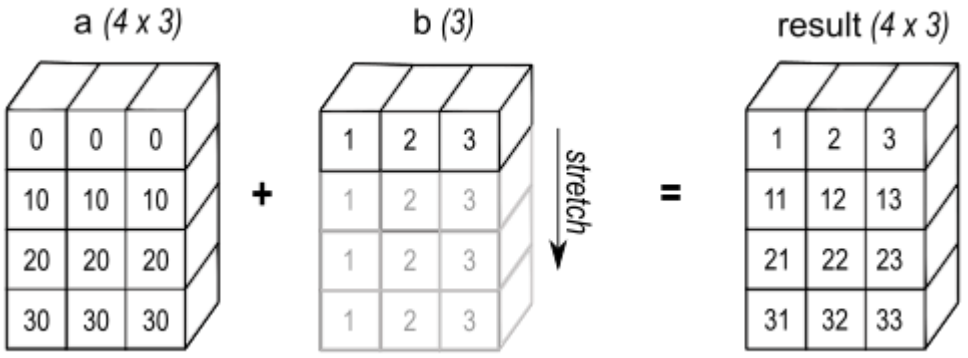
A (1D масив): 3
B (1D масив): 4 # виміри з кінця не відповідають

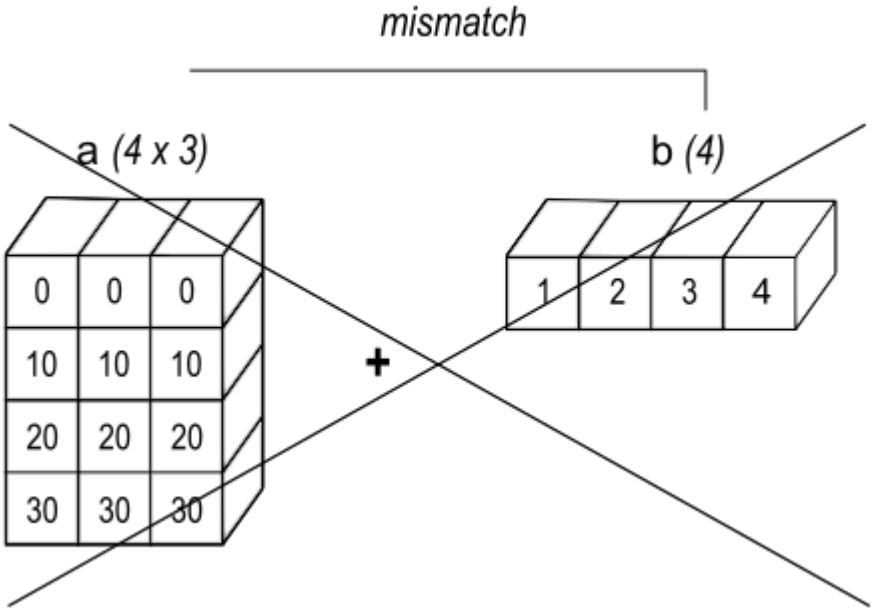
A (2D масив): 2 × 1
B (3D масив): 8 × 4 × 3 # друга з кінця вимірність не відповідає
Приклад broadcasting, коли 1D масив додається до 2D масиву:

```
a = np.array([[ 0.0,  0.0,  0.0],
[10.0, 10.0, 10.0],
[20.0, 20.0, 20.0],
[30.0, 30.0, 30.0]])
b = np.array([1.0, 2.0, 3.0])
a + b
array([[  1.,   2.,   3.],
[11.,  12.,  13.],
[21.,  22.,  23.],
[31.,  32.,  33.]])
```

```
b = np.array([1.0, 2.0, 3.0, 4.0])
a + b
```

Traceback (most recent call last):
ValueError: operands could not be broadcast together with shapes (4,3) (4,)
Як показано на Рисунку 2, b додається до кожного рядка a. На Рисунку 3 виникає виняток через несумісні форми.





Broadcasting надає зручний спосіб виконання зовнішнього добутку (або будь-якої іншої зовнішньої операції) двох масивів. Наступний приклад показує операцію зовнішнього додавання двох 1D масивів:

```
a = np.array([0.0, 10.0, 20.0, 30.0])
b = np.array([1.0, 2.0, 3.0])
a[:, np.newaxis] + b
array([[ 1., 2., 3.],
[11., 12., 13.],
[21., 22., 23.],
[31., 32., 33.]])
```