

---

# Hướng dẫn sử dụng NumPy

Phát hành 1.18.4

Được viết bởi cộng đồng NumPy



## NỘI DUNG

1 Thiết lập	3
2 Hướng dẫn bắt đầu nhanh	9
3 Điều cơ bản về NumPy	33
4 linh tinh	97
5 NumPy cho người dùng Matlab	103
6 Xây dựng từ nguồn	111
7 Sử dụng C-API NumPy	115
Chỉ mục mô-đun Python	163
Mục lục	165



Hướng dẫn này nhằm mục đích giới thiệu tổng quan về NumPy và giải thích cách cài đặt và sử dụng các tính năng quan trọng nhất của NumPy. Để biết tài liệu tham khảo chi tiết về các hàm và lớp có trong gói, hãy xem tài liệu tham khảo.



## ĐANG CÀI ĐẶT

## 1.1 NumPy là gì?

NumPy là gói cơ bản dành cho tính toán khoa học bằng Python. Đó là một thư viện Python cung cấp một đối tượng mảng đa chiều, các đối tượng dẫn xuất khác nhau (chẳng hạn như mảng và ma trận bị che) và một loạt các quy trình để thực hiện các thao tác nhanh trên mảng, bao gồm toán học, logic, thao tác hình dạng, sắp xếp, chọn, / 0, các phép biến đổi Fourier rời rạc, đại số tuyến tính cơ bản, các phép toán thống kê cơ bản, mô phỏng ngẫu nhiên và nhiều hơn nữa.

Cốt lõi của gói NumPy là đối tượng ndarray. Điều này gói gọn các mảng n chiều của các kiểu dữ liệu đồng nhất, với nhiều thao tác được thực hiện trong mã được biên dịch để tăng hiệu suất. Có một số khác biệt quan trọng giữa mảng NumPy và chuỗi Python tiêu chuẩn:

- Mảng NumPy có kích thước cố định khi tạo, không giống như danh sách Python (có thể phát triển linh hoạt). Thay đổi kích thước của ndarray sẽ tạo một mảng mới và xóa mảng gốc.
- Các phần tử trong mảng NumPy đều phải có cùng kiểu dữ liệu và do đó sẽ có cùng kích thước trong bộ nhớ. Ngoại lệ: người ta có thể có các mảng đối tượng (Python, bao gồm NumPy), do đó cho phép tạo các mảng có các phần tử có kích thước khác nhau.
- Mảng NumPy hỗ trợ các phép toán nâng cao và các loại phép tính khác trên số lượng lớn dữ liệu. Về cơ bản, các thao tác như vậy được thực thi hiệu quả hơn và sử dụng ít mã hơn mức có thể bằng cách sử dụng công cụ tích hợp sẵn của Python. trình tự.
- Ngày càng có nhiều gói dựa trên Python khoa học và toán học đang sử dụng mảng NumPy; mặc dù những thứ này thường hỗ trợ đầu vào theo trình tự Python, nhưng chúng chuyển đổi đầu vào đó thành mảng NumPy trước khi xử lý và chúng thường xuất ra mảng NumPy. Nói cách khác, để sử dụng hiệu quả nhiều (thậm chí là hầu hết) phần mềm dựa trên Python khoa học/toán học ngày nay, chỉ biết cách sử dụng các kiểu trình tự dựng sẵn của Python là không đủ - người ta cũng cần biết cách sử dụng mảng NumPy.

Các điểm về kích thước và tốc độ đặc biệt quan trọng trong tính toán khoa học. Một ví dụ đơn giản, hãy xem xét trường hợp nhân mỗi phần tử trong chuỗi 1-D với phần tử tương ứng trong một chuỗi khác có cùng độ dài. Nếu dữ liệu được lưu trữ trong hai danh sách Python, a và b, chúng ta có thể lặp lại từng phần tử:

```
c = []
for i in range(len(a)):
    c.append(a[i]*b[i])
```

Điều này tạo ra câu trả lời đúng, nhưng nếu mỗi a và b chứa hàng triệu số, chúng ta sẽ phải trả giá cho sự kém hiệu quả của việc lặp trong Python. Chúng ta có thể hoàn thành nhiệm vụ tương tự nhanh hơn nhiều trong C bằng cách viết (để rõ ràng, chúng ta bỏ qua việc khai báo và khởi tạo biến, phân bổ bộ nhớ, v.v.)

```
for (i = 0; i < hàng; i++):
    c[i] = a[i]*b[i];
,
```

Điều này giúp tiết kiệm cả chi phí liên quan đến việc diễn giải mã Python và thao tác với các đối tượng Python, nhưng lại phải trả giá bằng những lợi ích thu được từ việc mã hóa bằng Python. Hơn nữa, công việc mã hóa cần thiết sẽ tăng lên theo chiều của dữ liệu của chúng tôi. Ví dụ, trong trường hợp mảng 2-D, mã C (được rút gọn như trước) sẽ mở rộng thành

```
for (i = 0; i < hàng; i++) {
    for (j = 0; j < cột; j++) { c[i][j] =
        a[i][j]*b[i][j];
    }
}
```

NumPy mang đến cho chúng ta những điều tốt nhất của cả hai thế giới: các thao tác từng phần tử là "chế độ mặc định" khi có ndarray tham gia, nhưng thao tác từng phần tử được thực thi nhanh chóng bằng mã C được biên dịch trước. Trong NumPy

```
c = a * b
```

thực hiện những gì các ví dụ trước đó thực hiện, ở tốc độ gần C, nhưng với sự đơn giản về mã mà chúng tôi mong đợi từ một thứ dựa trên Python. Quả thực, thành ngữ NumPy thậm chí còn đơn giản hơn! Ví dụ cuối cùng này minh họa hai tính năng của NumPy là nền tảng cho phần lớn sức mạnh của nó: vector hóa và phát sóng.

### 1.1.1 Tại sao NumPy nhanh?

Vector hóa mô tả sự vắng mặt của bất kỳ vòng lặp, lập chỉ mục, v.v. rõ ràng nào trong mã - tất nhiên, những điều này đang diễn ra, chỉ là "đằng sau" trong mã C được biên dịch trước, được tối ưu hóa. Mã vectorized có nhiều ưu điểm, trong số đó là:

- mã vector hóa ngắn gọn hơn và dễ đọc hơn
- ít dòng mã hơn thường có nghĩa là ít lỗi hơn
- mã gần giống với ký hiệu toán học tiêu chuẩn hơn (thường làm cho việc mã hóa các cấu trúc toán học trở nên dễ dàng hơn)
- Vector hóa mang lại nhiều mã "Pythonic" hơn. Nếu không vector hóa, mã của chúng ta sẽ tràn ngập những lỗi không hiệu quả và khó đọc vòng lặp for.

Phát sóng là thuật ngữ được sử dụng để mô tả hành vi hoạt động ngầm định của từng phần tử; nói chung, trong NumPy, tất cả các phép toán, không chỉ các phép toán số học, mà cả các phép toán logic, khôn ngoan về bit, chức năng, v.v., đều hoạt động theo kiểu từng phần tử ngầm định này, tức là chúng phát sóng. Hơn nữa, trong ví dụ trên, a và b có thể là mảng nhiều chiều có cùng hình dạng, hoặc một mảng vô hướng và một mảng, hoặc thậm chí hai mảng có hình dạng khác nhau, miễn là mảng nhỏ hơn có thể "mở rộng" thành hình dạng của mảng lớn hơn. . . theo cách mà kết quả phát sóng là rõ ràng. Để biết "quy tắc" chi tiết về phát sóng, hãy xem [numpy.doc.broadcasting](#).

### 1.1.2 Ai còn sử dụng NumPy?

NumPy hỗ trợ đầy đủ cách tiếp cận hướng đối tượng, một lần nữa bắt đầu với ndarray. Ví dụ, ndarray là một lớp, sở hữu nhiều phương thức và thuộc tính. Nhiều phương thức của nó được phản ánh bởi các hàm trong không gian tên NumPy ngoài cùng, cho phép lập trình viên viết mã theo bất kỳ mô hình nào họ thích. Tính linh hoạt này đã cho phép phương ngữ mảng NumPy và lớp NumPy ndarray trở thành ngôn ngữ thực tế của trao đổi dữ liệu đa chiều được sử dụng trong Python.

## 1.2 Cài đặt NumPy

Trong hầu hết các trường hợp sử dụng, cách tốt nhất để cài đặt NumPy trên hệ thống của bạn là sử dụng gói dựng sẵn cho hệ điều hành của bạn. Vui lòng xem <https://scipy.org/install.html> để biết các liên kết đến các tùy chọn có sẵn.

Để biết hướng dẫn xây dựng gói nguồn, hãy xem [Xây dựng từ nguồn](#). Thông tin này hữu ích chủ yếu cho người dùng nâng cao.

## 1.3 Khắc phục sự cố ImportError

Lưu ý: Vì thông tin này có thể được cập nhật thường xuyên nên hãy đảm bảo bạn đang xem phiên bản cập nhật nhất.

### 1.3.1 Lỗi nhập khẩu

Trong một số trường hợp, sự cố cài đặt hoặc thiết lập không thành công có thể khiến bạn thấy thông báo lỗi sau:

QUAN TRỌNG: VUI LÒNG ĐỌC NÀY ĐỂ ĐƯỢC TƯ VẤN CÁCH GIẢI QUYẾT VẤN ĐỀ NÀY!

Nhập phần mở rộng c gọn gàng không thành công. Lỗi này có thể xảy ra vì nhiều lý do khác nhau, thường là do sự cố với quá trình thiết lập của bạn.

Lỗi cũng có thêm thông tin giúp bạn khắc phục sự cố:

- Phiên bản Python của bạn
- Phiên bản NumPy của bạn

Vui lòng kiểm tra cẩn thận cả hai điều này để xem liệu chúng có phải là điều bạn mong đợi hay không. Bạn có thể cần kiểm tra các biến môi trường PATH hoặc PYTHONPATH của mình (xem [Kiểm tra các biến môi trường](#) bên dưới).

Các phần sau liệt kê các sự cố thường được báo cáo tùy thuộc vào thiết lập của bạn. Nếu bạn có một vấn đề/giải pháp mà bạn nghĩ sẽ xuất hiện, vui lòng mở vấn đề NumPy để nó được thêm vào.

Có một số vấn đề thường được báo cáo tùy thuộc vào hệ thống/thiết lập của bạn. Nếu không có lời khuyên nào sau đây giúp ích cho bạn, hãy nhớ lưu ý những điều sau:

- bạn đã cài đặt Python như thế nào
- bạn đã cài đặt NumPy như thế nào
- hệ điều hành của bạn
- bạn có cài đặt nhiều phiên bản Python hay không
- nếu bạn xây dựng từ nguồn, các phiên bản trình biên dịch của bạn và lý tưởng nhất là nhặt ký xây dựng

khi điều tra thêm và yêu cầu hỗ trợ.

### Sử dụng Python từ conda (Anaconda)

Hãy đảm bảo rằng bạn đã kích hoạt môi trường conda của mình. Xem thêm hướng dẫn sử dụng conda.

### Sử dụng Anaconda/conda Python trong PyCharm

Có những vấn đề khá phổ biến khi sử dụng PyCharm cùng với Anaconda, vui lòng xem phần [hỗ trợ PyCharm](#)

### Raspberry Pi

Đôi khi có các sự cố được báo cáo về thiết lập Raspberry Pi khi cài đặt bằng cài đặt pip3 (hoặc cài đặt pip).

Những điều này thường sẽ đề cập đến:

```
libf77blas.so.3: không thể mở tệp đối tượng dùng chung : Không có tệp hoặc thư mục như vậy
```

Giải pháp sẽ là:

```
sudo apt-get cài đặt libatlas-base-dev
```

để cài đặt các thư viện còn thiếu mà NumPy tự biên dịch mong đợi (ATLAS có thể là nhà cung cấp đại số tuyến tính).

Hoặc sử dụng NumPy do Raspbian cung cấp. Trong trường hợp đó chạy:

```
pip3 gỡ cài đặt numpy # xóa phiên bản đã cài đặt trước đó apt install python3-numpy
```

### Gỡ lỗi bản dựng trên Windows

Thay vì xây dựng dự án của bạn ở chế độ GỠ LỖI trên windows, hãy thử xây dựng ở chế độ PHÁT HÀNH với các biểu tượng gỡ lỗi và không tối ưu hóa. Chế độ DEBUG đầy đủ trên windows thay đổi tên của các DLL mà python mong muốn tìm thấy, vì vậy nếu bạn muốn thực sự làm việc ở chế độ DEBUG, bạn sẽ cần phải biên dịch lại toàn bộ ngăn xếp mô-đun python mà bạn làm việc cùng, bao gồm cả NumPy.

#### Tắt cả các thiết lập

Đôi khi có thể có các sự cố đơn giản xảy ra với các bản cài đặt NumPy cũ hoặc kém. Trong trường hợp này, bạn có thể thử gỡ cài đặt và cài đặt lại NumPy. Đảm bảo rằng không tìm thấy NumPy sau khi gỡ cài đặt.

### Thiết lập phát triển

Nếu bạn đang sử dụng thiết lập phát triển, hãy đảm bảo chạy git clean -xdf để xóa tất cả các tệp không thuộc quyền kiểm soát phiên bản (hãy cẩn thận để không mất bất kỳ sửa đổi nào bạn đã thực hiện, ví dụ: site.cfg). Trong nhiều trường hợp, các tệp từ bản dựng cũ có thể dẫn đến bản dựng không chính xác.

### Kiểm tra các biến môi trường

Nói chung cách đặt và kiểm tra các biến môi trường tùy thuộc vào hệ thống của bạn. Nếu bạn có thể mở Shell python chính xác, bạn cũng có thể chạy phần sau trong python:

```
nhập os
PYTHONPATH = os.environ['PYTHONPATH'].split(os.pathsep) print(" PYTHONPATH là:", PYTHONPATH)
PATH = os.environ['PATH'].split(os.pathsep) print(" PATH là:", PATH)
```

Điều này chủ yếu có thể giúp ích cho bạn nếu bạn không chạy phiên bản python và/hoặc NumPy mà bạn muốn chạy.



---

chươnghai

---

## HƯỚNG DẪN BẮT ĐẦU NHANH

### 2.1 Điều kiện tiên quyết

Trước khi đọc hướng dẫn này, bạn nên biết một chút về Python. Nếu bạn muốn làm mới bộ nhớ của mình, hãy xem [hướng dẫn Python](#).

Nếu bạn muốn làm việc với các ví dụ trong hướng dẫn này, bạn cũng phải cài đặt một số phần mềm trên máy tính của mình. Vui lòng xem <https://scipy.org/install.html> để được hướng dẫn.

### 2.2 Những điều cơ bản

Đối tượng chính của NumPy là mảng đa chiều đồng nhất. Nó là một bảng gồm các phần tử (thường là số), tất cả cùng loại, được lập chỉ mục bởi một bộ số nguyên không âm. Trong kích thước NumPy được gọi là trực.

Ví dụ: tọa độ của một điểm trong không gian 3D [1, 2, 1] có một trực. Trục đó có 3 phần tử trong đó, vì vậy chúng ta nói nó có độ dài bằng 3. Trong ví dụ dưới đây, mảng có 2 trực. Trục thứ nhất có chiều dài 2, trục thứ hai có chiều dài 3.

```
[[ 1., 0., 0.],
 [ 0., 1., 2.]]
```

Lớp mảng của NumPy được gọi là ndarray. Nó còn được biết đến bởi mảng bí danh. Lưu ý rằng numpy.array không giống với lớp array.array trong Thư viện Python chuẩn, lớp này chỉ xử lý mảng một chiều và cung cấp ít chức năng hơn. Các thuộc tính quan trọng hơn của đối tượng ndarray là:

`ndarray.ndim` số trực (kích thước) của mảng.

`ndarray.shape` kích thước của mảng. Đây là một bộ số nguyên biểu thị kích thước của mảng trong mỗi kích thước. Đối với ma trận có n hàng và m cột, hình dạng sẽ là (n,m). Do đó, độ dài của bộ hình dạng là số trực, `ndim`.

`ndarray.size` tổng số phần tử của mảng. Điều này bằng với tích của các yếu tố hình dạng.

`ndarray.dtype` một đối tượng mô tả loại phần tử trong mảng. Người ta có thể tạo hoặc chỉ định các loại Python tiêu chuẩn đang sử dụng của dtype. Ngoài ra NumPy còn cung cấp các loại riêng. `numpy.int32`, `numpy.int16` và `numpy.float64` là một số ví dụ.

`ndarray.itemsize` kích thước tính bằng byte của từng phần tử của mảng. Ví dụ: một mảng gồm các phần tử thuộc loại `float64` có kích thước mục là 8 (=64/8), trong khi một mảng thuộc loại `complex32` có kích thước mục là 4 (=32/8). Nó tương đương với `ndarray.dtype.itemsize`.

`ndarray.data` bộ đệm chứa các phần tử thực tế của mảng. Thông thường chúng ta sẽ không cần sử dụng thuộc tính này bởi vì chúng ta sẽ truy cập các phần tử trong một mảng bằng cách sử dụng các phương tiện lập chỉ mục.

### 2.2.1 Một ví dụ

```
>>> nhập numpy dưới dạng np >>>
a = np.arange(15).reshape(3, 5)
>>> một
array([[ 0,  1,  2,  3,  4], [ 5,  6,  7,  8,
       9], [10, 11, 12, 13, 14]])
>>> a.shape ( 3 , 5 ) >>>
a.ndim

2
>>> a.dtype.name 'int64'

>>> a.itemsize
8
>>> a.kích thước
15
>>> type(a)
<type 'numpy.ndarray'> >>> b =
np.array([6, 7, 8]) >>> b

mảng([6, 7, 8]) >>>
type(b) <type
'numpy.ndarray'>
```

### 2.2.2 Tạo mảng

Có một số cách để tạo mảng.

Ví dụ: bạn có thể tạo một mảng từ danh sách hoặc bộ dữ liệu Python thông thường bằng cách sử dụng hàm `mảng`. Kiểu của mảng kết quả được suy ra từ kiểu của các phần tử trong chuỗi.

```
>>> nhập numpy dưới dạng np >>>
a = np.array([2,3,4])
>>> một
mảng([2, 3, 4]) >>>
a.dtype
dtype('int64') >>>
b = np.array([1.2, 3.5, 5.1]) >>> b.dtype
dtype('float64' ) )
```

Một lỗi thường gặp bao gồm việc gọi mảng với nhiều đối số là số, thay vì cung cấp một danh sách các số làm đối số.

```
a = np.array(1,2,3,4) >>> a =           # SAI >>>
np.array([1,2,3,4]) # ĐÚNG
```

Mảng biến đổi các chuỗi thành mảng hai chiều, chuỗi biến đổi các chuỗi thành mảng ba chiều, v.v.

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

Loại mảng cũng có thể được chỉ định rõ ràng tại thời điểm tạo:

```
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
mảng([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Thông thường, các phần tử của mảng ban đầu không xác định được nhưng kích thước của nó thì đã biết. Do đó, NumPy cung cấp một số chức năng để tạo mảng với nội dung giữ chỗ ban đầu. Những điều này giảm thiểu sự cần thiết của việc phát triển mảng, một hoạt động tốn kém.

Các số 0 của hàm tạo ra một mảng chứa đầy các số 0, các hàm số 1 tạo ra một mảng chứa đầy các số 1 và hàm trống tạo một mảng có nội dung ban đầu là ngẫu nhiên và phụ thuộc vào trạng thái của bộ nhớ. Theo mặc định, dtype của mảng được tạo là float64.

```
>>> np.zeros( (3,4) )
mảng([[0., 0., 0., 0.],
       [ 0., 0., 0., 0.],
       [ 0., 0., 0., 0.]])
>>> np.ones( (2,3,4), dtype=np.int16 ) array([[[ 1, 1, 1, 1],
                                                 [ 1, 1, 1, 1],
                                                 [[ 1, 1, 1, 1],
                                                 [ 1, 1, 1, 1],
                                                 [ 1, 1, 1, 1]],
                                                 dtype=int16)
>>> np.empty( (2,3) ) # chưa được khởi tạo, đều ra có thể thay đổi
mảng([[ 3.73603959e-262, 6.02658058e-154, 6.55490914e-260],
       [5.30498948e-313, 3.14673309e-307, 1.00000000e+000]])
```

Để tạo chuỗi số, NumPy cung cấp một hàm tương tự như phạm vi trả về mảng thay vì danh sách.

```
>>> np.arange( 10, 30, 5 )
mảng([10, 15, 20, 25])
>>> np.arange( 0, 2, 0.3 ) mảng([ 0.          # nó chấp nhận đối số float
, 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
```

Khi sắp xếp được sử dụng với các đối số dấu phẩy động, thường không thể dự đoán được số phần tử thu được, do đó chính xác của dấu phẩy động hữu hạn. Vì lý do này, tốt hơn nên sử dụng hàm linspace. Nhận làm đối số số phần tử mà chúng ta muốn, thay vì bước:

```
>>> từ số pi nhập numpy
>>> np.linspace( 0, 2, 9 ) mảng([ 0.          #9 số từ 0 đến 2
, 0.25, 0.5 , 0.75, 1. >>> x = np.linspace( 0, 2*pi, 100 ) , 1.25, 1.5, 1.75, 2. ])
· điểm # hữu ích để đánh giá chức năng ở nhiều
·
>>> f = np.sin(x)
```

Xem thêm:

mảng, số không, zeros\_like, những cái, one\_like, trống, trống\_like, arange, linspace, numpy.  
ngẫu nhiên.RandomState.rand, numpy.random.RandomState.randn, fromfunction, fromfile

### 2.2.3 Mảng in

Khi bạn in một mảng, NumPy hiển thị nó theo cách tương tự như các danh sách lồng nhau nhưng với bố cục sau:

- trực cuối cùng được in từ trái sang phải,
- từ thứ hai đến cuối cùng được in từ trên xuống dưới,
- phần còn lại cũng được in từ trên xuống dưới, mỗi lát cách nhau bằng một dòng trống.

Mảng một chiều sau đó được in dưới dạng hàng, mảng hai chiều dưới dạng ma trận và mảng ba chiều dưới dạng danh sách ma trận.

```
>>> a = np.arange(6) >>>                               #mảng #1d
print(a)
[0 1 2 3 4 5]
'
>>> b = np.arange(12).reshape(4,3) >>>                 # mảng 2d
print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
'
>>> c = np.arange(24).reshape(2,3,4) >>>             #mảng 3d
print(c)
[[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11 ]]
 [[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]]
```

Xem bên dưới để biết thêm chi tiết về việc định hình lại.

Nếu một mảng quá lớn không thể in được, NumPy sẽ tự động bỏ qua phần trung tâm của mảng và chỉ in phần góc:

```
>>> print(np.arange(10000))
,    0      1      2 ,    9997 9998 9999]
'
>>> print(np.arange(10000).reshape(100,100))
[[ 1      0      2 ,    97      98      99]
 [ 100   101   102 ,    197   198   199]
 [ 200   201   202 ,    297   298   299]
 ,
 [9700 9701 9702 ,    9797 9798 9799]
 [9800 9801 9802 ,    9897 9898 9899]
 [9900 9901 9902 ,    9997 9998 9999]]
```

Để tắt hành vi này và buộc NumPy in toàn bộ mảng, bạn có thể thay đổi tùy chọn in bằng cách sử dụng `set_printoptions`.

```
>>> np.set_printoptions(ngưỡng=sys.maxsize)                      # mô-đun sys nên được nhập
```

## 2.2.4 Các thao tác cơ bản

Các toán tử số học trên mảng áp dụng theo từng phần tử. Một mảng mới được tạo và chứa kết quả.

```
>>> a = np.array([20,30,40,50]) >>> b
= np.arange(4) >>> b

mảng([0, 1, 2, 3]) >>> c
= ab
>>> c
mảng([20, 29, 38, 47]) >>> b**2

mảng([0, 1, 4, 9]) >>
10*np.sin(a)
mảng([ 9.12945251, -9.88031624, 7.4511316 >>> a<35           , -2.62374854])

mảng([ Đúng, Đúng, Sai, Sai])
```

Không giống như trong nhiều ngôn ngữ ma trận, toán tử tích \* hoạt động theo từng phần tử trong mảng NumPy. Tích ma trận có thể được thực hiện bằng toán tử @ (trong python >=3.5) hoặc hàm hoặc phương thức dấu chấm:

```
>>> A = np.array([[1,1],
'                 [0,1]] )
>>> B = np.array([[2,0],
'                 [3,4]] )
>>> A * B
# sản phẩm theo nguyên tố
mảng([[2, 0],
'     [0, 4]])
>>> Mảng A
# sản phẩm ma trận
@ B ([[5, 4], [3,
4]])
>>> Mảng
# một sản phẩm ma trận khác
A.dot(B) ([[5, 4],
[3, 4]])
```

Một số thao tác, chẳng hạn như += và \*=, hoạt động tại chỗ để sửa đổi một mảng hiện có thay vì tạo một mảng mới.

```
>>> a = np.ones((2,3), dtype=int) >>> b
= np.random.random((2,3)) >>> a *= 3

>>> một
mảng([[3, 3, 3], [3,
3, 3]]) >>>
b += a
>>> b
array([[ 3.417022      , 3.72032449, 3.00011437],
       [ 3.30233257, 3.14675589, 3.09233859]]) # b không được tự động
>>> a += b
# chuyển đổi thành kiểu số nguyên
Traceback (cuộc gọi gần đây nhất):
'
TypeError: Không thể truyền ufunc thêm đầu ra từ dtype('float64') sang dtype('int64') với ` quy tắc truyền 'same_kind'
```

Khi thao tác với các mảng thuộc nhiều loại khác nhau, loại của mảng kết quả sẽ tương ứng với loại tổng quát hơn hoặc chính xác hơn (một hành vi được gọi là upcasting).

```
>>> a = np.ones(3, dtype=np.int32) >>> b
= np.linspace(0,np.pi,3)
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
>>> b.dtype.name
'float64'
>>> c = a+b
>>> c
mảng([ 1. , 2.57079633, 4.14159265])
>>> c.dtype.name
'float64'
>>> d = np.exp(c*1j) >>>
d
mảng([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```

Nhiều phép toán một ngôi, chẳng hạn như tính tổng của tất cả các phần tử trong mảng, được triển khai như các phương thức của lớp ndarray.

```
>>> a = np.random.random((2,3))
>>> một
mảng([[0.18626021, 0.34556073, 0.39676747],
       [0.53881673, 0.41919451, 0.6852195]]) >>>
a.sum()
2.5718191614547998
>>> a.min()
0.1862602113776709
>>> a.max()
0.6852195003967595
```

Theo mặc định, các thao tác này áp dụng cho mảng như thể nó là một danh sách các số, bất kể hình dạng của nó như thế nào. Tuy nhiên, bằng cách chỉ định tham số trực, bạn có thể áp dụng một thao tác đọc theo trực đã chỉ định của một mảng:

```
>>> b = np.arange(12).reshape(3,4)
>>> b
mảng([[ 0, 1, 2, 3], [ 4, 5, 6,
       7], [ 8, 9, 10, 11]])

,
>>> b.sum(axis=0) # tổng của mỗi cột
array([12, 15, 18, 21])
,
>>> b.min(axis=1) # phút mỗi hàng
array([0, 4, 8])
,
>>> b.cumsum(axis=1) # tổng tích lũy đọc theo mỗi hàng
array([[ 0, 1, 3, 6], [ 4, 9,
       15, 22], [ 8, 17,
       27, 38]])
```

## 2.2.5 Hàm phổ quát

NumPy cung cấp các hàm toán học quen thuộc như sin, cos và exp. Trong NumPy, chúng được gọi là "phổ quát chức năng"(ufunc). Trong NumPy, các hàm này hoạt động theo từng phần tử trên một mảng, tạo ra một mảng làm đầu ra.

```
>>> B = np.arange(3)
>>> B
mảng([0, 1, 2])
>>> np.exp(B)
mảng([ 1.          ,  2.71828183,  7.3890561 ])
>>> np.sqrt(B)
mảng ([ 0.          ,  1.          ,  1.41421356])
>>> C = np.array([2., -1., 4.])
>>> np.add(B, C)
mảng([ 2.,  0.,  6.])
```

Xem thêm:

tất cả, bất kỳ, apply\_along\_axis, argmax, argmin, argsort, Average, bincount, ceil, clip, conj, corrcoef, cov, cross, cumprod, cumsum, diff, dot, sàn, bên trong, inv, lexsort, max, tối đa, trung bình, trung vị, tối thiểu, tối thiểu, khác không, bên ngoài, sản phẩm, lại, tròn, sắp xếp, tiêu chuẩn, tính tổng, theo dõi, hoán vị, var, vdot, vector hóa, ở đâu

## 2.2.6 Lập chỉ mục, cắt và lặp

Mảng một chiều có thể được lập chỉ mục, cắt lát và lặp lại, giống như [danh sách](#) và các chuỗi Python khác.

```
>>> a = np.arange(10)**3
>>> một
mảng([ 0, >>      1,      8, 27, 64, 125, 216, 343, 512, 729])
a[2]
...
>>> a[2:5]
mảng([ 8, 27, 64])
>>> a[:6:2] = -1000 # tương đương với a[0:6:2] = -1000; từ đầu đến vị trí 6,
    quyển, đặt mọi phần tử thứ 2 thành -1000
    ↵
>>> một
array([-1000,      1, -1000,      27, -1000, 125,      216, 343, 512,      729])
>>> a[ : :-1]
array([ 729, 512, 343, >>> for i      216,      125, -1000,      27, -1000,      1, -1000])
in a:
    ↵
        in(i**(1/3.))
    ↵
nan
1.0
nan
3.0
nan
5.0
6.0
7.0
8.0
9.0
```

Mảng đa chiều có thể có một chỉ mục trên mỗi trục. Các chỉ số này được đưa ra trong một bộ dữ liệu được phân tách bằng dấu phẩy:

```
>>> định nghĩa f(x,y):
'         trả về 10*x+y
'
>>> b = np.fromfunction(f,(5,4),dtype=int)
>>> b
mảng([[0, 1, 2, 3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
>>> b[0:5, 1]                                     # mỗi hàng trong cột thứ hai của b
mảng([ 1, 11, 21, 31, 41])
>>> b[ : ,1]                                     # tương đương với ví dụ trước
mảng([ 1, 11, 21, 31, 41])
>>> b[1:3, : ]                                    # mỗi cột ở hàng thứ hai và thứ ba của b
mảng([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Khi cung cấp ít chỉ mục hơn số lượng trực, các chỉ mục bị thiếu được coi là các lát cắt hoàn chỉnh:

```
>>> b[-1]                                         # hàng cuối cùng. Tương đương với b[-1,:]
mảng([40, 41, 42, 43])
```

Biểu thức trong ngoặc trong  $b[i]$  được coi là  $i$  theo sau là nhiều trường hợp : nếu cần để biểu thị các trực còn lại. NumPy cũng cho phép bạn viết điều này bằng cách sử dụng dấu chấm dưới dạng  $b[i,...]$ .

Các dấu chấm (...) biểu thị số lượng dấu hai chấm cần thiết để tạo ra một bộ chỉ mục hoàn chỉnh. Ví dụ: nếu  $x$  là một mảng có 5 trực thì

- $x[1,2,...]$  tương đương với  $x[1,2,:,:,:]$ ,
- $x[...,3]$  đến  $x[:, :, :, :, 3]$  và
- $x[4,...,5,:]$  đến  $x[4, :, :, 5, :]$ .

```
>>> c = np.array( [[[ 0, 1, 2],
                   [ 10, 12, 13]],
                   [[100,101,102],
                   [110,112,113]]])
>>> c.hình dạng
(2, 2, 3)
>>> c[1,...]                                       # giống như c[1,:,:] hoặc c[1]
mảng([[100, 101, 102],
       [110, 112, 113]])
>>> c[...,2]                                       # giống như c[:, :, 2]
mảng([[ 2, 13],
       [102, 113]])
```

Việc lặp lại các mảng đa chiều được thực hiện đối với trực đầu tiên:

```
>>> cho hàng trong b:
'         in (hàng)
'
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
[30 31 32 33]
[40 41 42 43]
```

Tuy nhiên, nếu muốn thực hiện một thao tác trên từng phần tử trong mảng, người ta có thể sử dụng thuộc tính `flat` là một iterator . Trên tất cả các phần tử của mảng:

```
>>> cho phần tử trong b.flat:
    in (phần tử)

0
1
2
3
10
11
12
13
20
21
22
23
30
31
32
33
40
41
42
43
```

Xem thêm:

[Lập chỉ mục, arrays.indexing](#) (tham khảo), `newaxis`, `ndenumerate`, [chỉ mục](#)

## 2.3 Thao tác hình dạng

### 2.3.1 Thay đổi hình dạng của mảng

Một mảng có hình dạng được cho bởi số phần tử đọc theo mỗi trực:

```
>>> a = np.floor(10*np.random.random((3,4)))
>>> một
mảng([[ 2.,  8.,  0.,  6.], [ 4.,  5.,
       1.,  1.], [ 8.,  9.,  3.,  6.]])
```

```
>>> a.hình
(3, 4)
```

Hình dạng của một mảng có thể được thay đổi bằng nhiều lệnh khác nhau. Lưu ý rằng ba lệnh sau đều trả về một mảng đã sửa đổi, nhưng không thay đổi mảng ban đầu:

```
>>> a.ravel() # trả về mảng, mảng phẳng([ 2.,  8.,  0.,  6.,  4.,  5.,  1.,
       1.,  8.,  9.,  3.,  6.]) >>> a.reshape(6,2) # trả về mảng có hình dạng đã được sửa đổi
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
mảng([[ 2., 8.], [ 0., 6.],
      [ 4., 5.], [ 1.,
      1.], [ 8., 9.],
      [ 3., 6. ]])
>>> aT # trả về
mảng, chuyển
mảng([[ 2., 4., 8.], [ 8., 5., 9.], [ 0., 1., 3.], [ 6., 1.,
6.]])
```

```
>>> aHình dạng (4,
3)
>>> a.hình (3, 4)
```

Thứ tự các phần tử trong mảng do `ravel()` thường là “kiểu C”, tức là chỉ số ngoài cùng bên phải “thay đổi nhanh nhất”, nên phần tử sau  $a[0,0]$  là  $a[0,1]$ . Nếu mảng được định hình lại thành một số hình dạng khác thì mảng đó lại được coi là “kiểu C”. NumPy thường tạo các mảng được lưu trữ theo thứ tự này, vì vậy `ravel()` thường sẽ không sao chép đổi số của nó, nhưng nếu mảng được tạo bằng cách lấy các lát của mảng khác hoặc được tạo bằng các tùy chọn bất thường, thì mảng đó có thể cần phải được sao chép.

Các hàm `ravel()` và `reshape()` cũng có thể được hướng dẫn bằng cách sử dụng một đối số tùy chọn để sử dụng mảng kiểu FORTRAN, trong đó chỉ mục ngoài cùng bên trái thay đổi nhanh nhất.

Hàm định hình lại trả về đối số của nó với hình dạng đã được sửa đổi, trong khi phương thức `ndarray.resize` sẽ sửa đổi chính mảng đó:

```
>>> một
mảng([[ 2., 8., 0., 6.], [ 4., 5., 1., 1.],
      [ 8., 9., 3., 6.]]) >>>
a.resize((2,6))

>>> một
mảng([[ 2., 8., 0., 6., 4., 5.],
      [ 1., 1., 8., 9., 3., 6.]])
```

Nếu một thứ nguyên được cho là -1 trong thao tác định hình lại thì các thứ nguyên khác sẽ được tính toán tự động:

```
>>> a.reshape(3,-1)
array([[ 2., 8., 0., 6.], [ 4., 5., 1.,
1.], [ 8., 9., 3., 6.]])
```

Xem thêm:

`ndarray.shape`, định hình lại, thay đổi kích thước, chia nhỏ

### 2.3.2 Xếp chồng các mảng khác nhau

Một số mảng có thể được xếp chồng lên nhau dọc theo các trục khác nhau:

```
>>> a = np.floor(10*np.random.random((2,2)))
>>> một
mảng([[ 8.,  8.], [ 0.,  0.]])  
  
>>> b = np.floor(10*np.random.random((2,2))) >>> b  
  
mảng([[ 1.,  8.], [ 0.,  4.]])
>>> np.vstack((a,b))
array([[ 8.,  8.], [ 0.,  0.],
[ 1.,  8.], [ 0.,  4.]]) >>>
np.hstack((a,b))
array([[ 8.,  8.,
 1.,  8.], [ 0.,  0.,
 0.,  4.]])
```

Hàm `columns_stack` xếp các mảng 1D dưới dạng cột thành mảng 2D. Nó chỉ tương đương với `hstack` đối với mảng 2D:

```
>>> from numpy import newaxis >>>
np.column_stack((a,b)) array([[ 8.,  8.,  1.,
8.], [ 0.,  0.,  0.,  4.]]) >>> a = np.array([4.,2.])
>>> b = np.array([3.,8.]) >>>
np.column_stack((a,b)) mảng ([[ 4.,  3.,
2.,  8.]]) >>> np.hstack((a,b)) array([ 4.,
2.,  3.,  8.]) >>> a[:,newaxis] mảng([[ 4.,
2.]])  
  
# kết quả lại khác  
  
# điều này cho phép có vectơ cột 2D  
  
>>> np.column_stack((a[:,newaxis],b[:,newaxis])) array([[ 4.,  3.], [ 2.,  8.]])  
  
>>> np.hstack((a[:,newaxis],b[:,newaxis])) # kết quả là cùng một mảng([[ 4.,  3.], [ 2.,  8.]])
```

Mặt khác, hàm `row_stack` tương đương với `vstack` đối với bất kỳ mảng đầu vào nào. Nói chung, đối với các mảng có nhiều hơn hai chiều, `hstack` xếp chồng dọc theo trục thứ hai của chúng, `vstack` xếp chồng dọc theo trục đầu tiên của chúng và ghép nối cho phép một đối số tùy chọn đưa ra số trục dọc theo đó việc ghép nối sẽ xảy ra.

#### Ghi chú

Trong trường hợp phức tạp, `r_` và `c_` rất hữu ích cho việc tạo mảng bằng cách xếp chồng các số dọc theo một trục. Chúng cho phép sử dụng các ký tự có phạm vi ":"

```
>>> np.r_[1:4,0,4] mảng([1, 2,
3, 0, 4])
```

Khi được sử dụng với mảng làm đối số, `r_` và `c_` tương tự như `vstack` và `hstack` trong hành vi mặc định của chúng, nhưng cho phép một đối số tùy chọn đưa ra số trực cần nối.

Xem thêm:

`hstack`, `vstack`, `cot_stack`, `nối`, `c_`, `r_`

### 2.3.3 Chia một mảng thành nhiều mảng nhỏ hơn

Sử dụng `hsplit`, bạn có thể chia một mảng dọc theo trục ngang của nó, bằng cách chỉ định số lượng mảng có hình dạng bằng nhau cần trả về hoặc bằng cách chỉ định các cột mà sau đó việc phân chia sẽ diễn ra:

```
>>> a = np.floor(10*np.random.random((2,12)))
>>> một
mảng([[ 9., 5., 6., 3., 6., 8., 0., 7., 9., 7., 2., 7.],
       [ 1., 4., 9., 2., 2., 1., 0., 6., 2., 2., 4., 0.]])
>>> np.hsplit(a,3) # Chia a thành 3 [array([[ 9.,
5., 6., 3.], [ 1., 4., 9., 2.]]),
mảng([[ 6., 8., 0., 7.], [ 2., 1., 0., 6.]]), mảng([[ 9., 7., 2.,
7.], [ 2., 2., 4., 0.]]]

>>> np.hsplit(a,(3,4)) # Tách a sau cột thứ ba và cột thứ tư [array([[ 9., 5., 6.], [ 1., 4.,
9. ]]), mảng([[ 3.], [ 2.]]),
mảng([[ 6., 8., 0., 7., 9., 7., 2.,
7.]], [ 2 . , 1., 0., 6., 2., 4., 0.]]]
```

`vsplit` phân chia dọc theo trục tung và `array_split` cho phép người ta chỉ định dọc theo trục nào sẽ phân chia.

## 2.4 Bản sao và lượt xem

Khi vận hành và thao tác với mảng, dữ liệu của chúng đôi khi được sao chép sang một mảng mới và đôi khi không. Đây thường là nguồn gây nhầm lẫn cho người mới bắt đầu. Có ba trường hợp:

### 2.4.1 Không có bản sao nào cả

Các phép gán đơn giản không tạo ra bản sao của các đối tượng mảng hoặc dữ liệu của chúng.

```
>>> a = np.arange(12) >>>
b = a                      # không có đối tượng mới nào được tạo
>>> b là a                  # a và b là hai tên của cùng một đối tượng ndarray
ĐÚNG VẬY
>>> b.shape = 3,4            # thay đổi hình dạng của một
>>> a.shape
(3, 4)
```

Python chuyển các đối tượng có thể thay đổi làm tham chiếu, vì vậy các lệnh gọi hàm không tạo ra bản sao.

```
>>> định nghĩa f(x):
'      in(id(x))
'
>>> id(a)                   # id là mã định danh duy nhất của một đối tượng
148293216
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
>>> f(a)
148293216
```

### 2.4.2 Xem hoặc sao chép nông

Các đối tượng mảng khác nhau có thể chia sẻ cùng một dữ liệu. Phương thức xem tạo ra một đối tượng mảng mới trông giống như vậy dữ liệu.

```
>>> c = a.view()
>>> c là một
SAI
>>> c.base là một                                # c là chế độ xem dữ liệu thuộc sở hữu của một
ĐÚNG VẬY
>>> c.flags.owndata
SAI
'
>>> c.shape = 2,6 >>>                               # hình dạng của a không thay đổi
a.shape
(3, 4)
>>> c[0,4] = 1234                                    # a thay đổi dữ liệu
>>> một
mảng([[ 0, [1234,          1,          2,          3],
       [ 8,          5,          7],
       9,          6, 10,         11]])
```

Cắt một mảng trả về chế độ xem của nó:

```
>>> s = a[:, 1:3]           # khoảng trắng được thêm vào để rõ ràng; cũng có thể được viết "s = a[:, 1:3]"
>>> s[:] = 10               # s[:] là chế độ xem của s. Lưu ý sự khác biệt giữa s=10 và
   s[:]=10
>>> một
mảng([[ 0, [1234,          10,          10,          3],
       [ 8,          10,          10,          7],
       10,          10,         11]])
```

### 2.4.3 Bản sao sâu

Phương thức sao chép tạo một bản sao hoàn chỉnh của mảng và dữ liệu của nó.

```
>>> d = a.copy()                      # một đối tượng mảng mới với dữ liệu mới là
   đã tạo
>>> d là một
SAI
>>> d.base là một                    # d không chia sẻ bất cứ điều gì với a
SAI
>>> d[0,0] = 999
>>> một
mảng([[ 0, [1234,          10,          10,          3],
       [ 8,          10,          10,          7],
       10,          10,         11]])
```

Đôi khi bản sao nên được gọi sau khi cắt nếu mảng ban đầu không còn cần thiết nữa. Ví dụ, giả sử `a` là kết quả trung gian rất lớn và kết quả cuối cùng `b` chỉ chứa một phần nhỏ của `a`, nên tạo một bản sao sâu

khi xây dựng b bằng cách cắt:

```
>>> a = np.arange(int(1e8)) >>> b =
a[:100].copy() >>> del a # bộ nhớ
của ``a`` có thể được giải phóng.
```

Nếu `b = a[:100]` được sử dụng thay thế, `a` được tham chiếu bởi `b` và sẽ tồn tại trong bộ nhớ ngay cả khi `del a` được thực thi.

#### 2.4.4 Tổng quan về hàm và phương thức

Dưới đây là danh sách một số tên phương thức và hàm NumPy hữu ích được sắp xếp theo danh mục. Xem các thói quen để biết danh sách đầy đủ.

Tạo mảng `arange`, `mảng`, sao chép, `trống`, `trống_like`, `mắt`, `fromfile`, `fromfunction`, `Identity`, `linspace`, `logspace`, `mggrid`, `ogrid`, `one`, `one_like`, `r`, `zeros`, `zeros_like`

Chuyển đổi `ndarray.astype`, `less_1d`, `less_2d`, `less_3d`, `mat`

Thao tác `array_split`, `cột_stack`, ghép nối, đường chéo, `dsplit`, `dstack`, `hsplit`, `hstack`, `ndarray.item`, `newaxis`, `ravel`, lặp lại, định hình lại, thay đổi kích thước, bóp, hoán đổi, lấy, chuyển vị, `vsplit`, `vstack`

Tất cả, bất kỳ, khác không, ở đâu

Thứ tự `argmax`, `argmin`, `argsort`, `max`, `phút`, `ptp`, `sắp xếp tìm kiếm`, `sắp xếp`

Các thao tác chọn, nén, `cumprod`, `cumsum`, bên trong, `ndarray.fill`, `imag`, `prod`, `put`, `putmask`, thực, tổng

Thống kê cơ bản `cov,mean,std,var`

Đại số tuyến tính cơ bản chéo, dấu chấm, bên ngoài, `linalg.svd`, `vdot`

## 2.5 Ít cơ bản hơn

### 2.5.1 Quy tắc phát sóng

Việc phát sóng cho phép các chức năng phô quát xử lý một cách có ý nghĩa với các đầu vào không có hình dạng giống nhau.

Quy tắc phát sóng đầu tiên là nếu tất cả các mảng đầu vào không có cùng số chiều, thì số "1" sẽ được thêm vào trước hình dạng của các mảng nhỏ hơn cho đến khi tất cả các mảng có cùng số chiều.

Quy tắc phát sóng thứ hai đảm bảo rằng các mảng có kích thước bằng 1 đọc theo một chiều cụ thể sẽ hoạt động như thể chúng có kích thước bằng mảng có hình dạng lớn nhất đọc theo chiều đó. Giá trị của phần tử mảng được giả định là giống nhau đọc theo chiều đó đối với mảng "phát sóng".

Sau khi áp dụng quy tắc phát sóng, kích thước của tất cả các mảng phải khớp nhau. Thông tin chi tiết có thể được tìm thấy trong [Truyền phát rộng](#).

## 2.6 Lập chỉ mục và thủ thuật lập chỉ mục ưa thích

Numpy cung cấp nhiều phương tiện lập chỉ mục hơn các chuỗi Python thông thường. Ngoài việc lập chỉ mục theo số nguyên và lát cắt, như chúng ta đã thấy trước đây, mảng có thể được lập chỉ mục bởi mảng số nguyên và mảng boolean.

#### 2.6.1 Lập chỉ mục với mảng chỉ số

```

>>> a = np.arange(12)**2 >>>
i = np.array( [ 1,1,3,8,5 ] >>> a[i]      ,
mảng([ 1, 1, >>>      9, 64, 25])           #12 số bình phương đầu tiên
                                                # một loạt các chỉ số
                                                # các phần tử của a tại vị trí i

>>> j = np.array( [ >>>  [ 3, 4], [ 9, 7 ]      , ,           # mảng chỉ số hai chiều
a[j]                                         # hình dạng giống như j
array([[ 9, 16,
       [81, 49]])
```

Khi mảng được lập chỉ mục a là nhiều chiều, một mảng chỉ số duy nhất sẽ tham chiếu đến chiều đầu tiên của a. Sau đây ví dụ này cho thấy hành vi này bằng cách chuyển đổi hình ảnh nhãn thành hình ảnh màu bằng bảng màu.

Chúng ta cũng có thể đưa ra các chỉ mục cho nhiều hơn một chiều. Màng chỉ số cho mỗi chiều phải giống nhau

```
>>> a = np.arange(12).reshape(3,4)
>>> mảng
mảng([[0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])
>>> i = np.array( [ [0,1],
                   [1,2] ]      ,           # chỉ số cho độ mờ đầu tiên của
,
                   [2,1],           # chỉ số cho độ mờ thứ hai
,
                   [3,3] ]      ,
,
>>> a[i,j]                      # i và j phải có hình dạng bằng nhau
mảng([[ 2, 5],
       [7, 11]])
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
>>> a[i,2]
mảng([[2, 6,
       [ 6, 10]]),
'
>>> a[:,j] #tức là, một[ : , j]
mảng([[ 2, 1],
       [ 3, 3],
       [[ 6, 5],
        [ 7, 7]],
       [[10, 9],
        [11, 11]]])
```

Đương nhiên, chúng ta có thể đặt i và j trong một chuỗi (chẳng hạn như một danh sách) và sau đó thực hiện lập chỉ mục cho danh sách đó.

```
>>> l = [i,j]
>>> a[l] #tương đương với a[i,j]
mảng([[ 2, 5],
       [7, 11]])
```

Tuy nhiên, chúng ta không thể làm điều này bằng cách đặt i và j vào một mảng, vì mảng này sẽ được hiểu là lập chỉ mục cho chiều thứ nhất của a.

```
>>> s = np.array( [i,j] )
>>> mảng [s] # không phải điều chúng ta muốn
Traceback (cuộc gọi gần đây nhất):
Tệp "<stdin>", dòng 1, ở định dạng ?
IndexError: chỉ mục (3) nằm ngoài phạm vi (0<=index<=2) trong thứ nguyên 0
'
>>> a[(các) b] # giống như a[i,j]
mảng([[ 2, 5],
       [7, 11]])
```

Một cách sử dụng phổ biến khác của việc lập chỉ mục với mảng là tìm kiếm giá trị lớn nhất của chuỗi phụ thuộc vào thời gian:

```
>>> time = np.linspace(20, 145, 5) >> #thang thời gian
data = np.sin(np.arange(20)).reshape(5,4) >> thời gian #4 loạt phụ thuộc vào thời gian

mảng([ 20. >>      ,51,25,           82,5    , 113,75, 145. ])
dữ liệu
mảng([[ 0. , 0.84147098, 0.90929743, 0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 0.6569866 ],
       [0,98935825, 0,41211849, -0,54402111, -0,99999021],
       [-0,53657292, 0,42016704, 0,99060736, 0,65028784],
       [-0,28790332, -0,96139749, -0,75098725, 0,14987721]])
'
>>> ind = data.argmax(axis=0) >> # chỉ số cực đại của mỗi chuỗi
ind
mảng([2, 0, 3, 1])
'
>>> time_max = thời gian[ind] # lần tương ứng với cực đại
'
>>> data_max = data[ind, range(data.shape[1])] # => data[ind[0],0], data[ind[1],1]...
'
>>> thời gian_max
mảng([ 82.5 >>      , 20.     , 113,75, 51,25])
data_max
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
mảng([ 0,98935825, 0,84147098, 0,99060736, 0,6569866 ])
'
>>> np.all(data_max == data.max(axis=0))
Đúng vậy
```

Bạn cũng có thể sử dụng tính năng lặp chỉ mục với mảng làm mục tiêu để gán cho:

```
>>> a = np.arange(5)
>>> một
mảng([0, 1, 2, 3, 4]) >>
a[[1,3,4]] = 0
>>> một
mảng([0, 0, 2, 0, 0])
```

Tuy nhiên, khi danh sách các chỉ mục chứa các lần lặp lại, việc gán được thực hiện nhiều lần, để lại giá trị cuối cùng:

```
>>> a = np.arange(5) >>
a[[0,0,2]]=1,2,3
>>> một
mảng([2, 1, 3, 3, 4])
```

Điều này đủ hợp lý, nhưng hãy cẩn thận nếu bạn muốn sử dụng cấu trúc `+=` của Python, vì nó có thể không thực hiện những gì bạn mong đợi:

```
>>> a = np.arange(5) >>
a[[0,0,2]]+=1
>>> một
mảng([1, 1, 3, 3, 4])
```

Mặc dù 0 xuất hiện hai lần trong danh sách chỉ mục, phần tử thứ 0 chỉ được tăng một lần. Điều này là do Python yêu cầu "`a+=1`" tương đương với "`a = a + 1`".

## 2.6.2 Lập chỉ mục với mảng Boolean

Khi chúng tôi lập chỉ mục các mảng chỉ mục (số nguyên), chúng tôi sẽ cung cấp danh sách các chỉ mục để chọn. Với các chỉ số boolean, cách tiếp cận sẽ khác; chúng tôi chọn rõ ràng mục nào trong mảng chúng tôi muốn và mục nào chúng tôi không muốn.

Cách tự nhiên nhất mà người ta có thể nghĩ đến để lập chỉ mục boolean là sử dụng các mảng boolean có hình dạng giống như mảng ban đầu:

```
>>> a = np.arange(12).reshape(3,4) >> b
= a > 4
>>> b
# b là một boolean có hình dạng a
mảng([[Sai, Sai, Sai, Sai],
       [Sai, Đúng, Đúng, Đúng],
       [Đúng, Đúng, Đúng, Đúng]]) >> a[b]
Mảng #1d với các phần tử được chọn
mảng([ 5, 6,           7, 8, 9, 10, 11])
```

Thuộc tính này có thể rất hữu ích trong các bài tập:

```
>>> a[b] = 0
`   trở thành 0
>>> một
mảng([[0, 1, 2, 3],
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

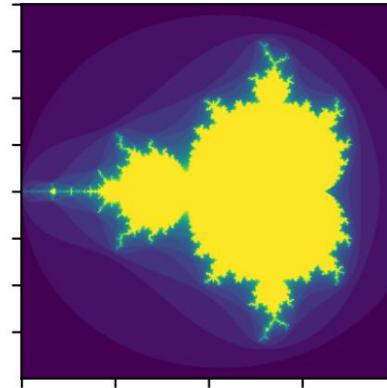
```
[4, 0, 0, 0],
[0, 0, 0, 0])
```

Bạn có thể xem ví dụ sau để biết cách sử dụng tính năng lập chỉ mục boolean để tạo ra hình ảnh của tập Mandelbrot:

```
>>> nhập numpy dưới dạng np
>>> nhập matplotlib.pyplot dưới dạng plt
>>> def mandelbrot( h,w, maxit=20 ):
    """Trả về hình ảnh của fractal Mandelbrot có kích thước (h,w)."""
    y,x = np.ogrid[ -1.4:1.4:h*1j, -2:0.8:w*1j ]
    c = x+y*1j
    z = c
    divtime = maxit + np.zeros(z.shape, dtype=int)

    cho tôi trong phạm vi (maxit):
        z = z**2 + c
        phân kỳ = z*np.conj(z) > 2**2           #ai đang chia rẽ
        div_now = phân kỳ & (divtime==maxit) # ai hiện đang phân kỳ
        divtime[div_now] =                   # lưu ý khi nào
        iz[phân kỳ] = 2                     #tránh phân kỳ quá nhiều

    trả lại thời gian phân chia
>>> plt.imshow(mandelbrot(400,400))
>>> plt.show()
```



Cách lập chỉ mục thứ hai bằng boolean tương tự như lập chỉ mục số nguyên hơn; đối với mỗi chiều của mảng chúng ta đưa ra một mảng boolean 1D chọn các lát mà chúng ta muốn:

```
>>> a = np.arange(12).reshape(3,4)
>>> b1 = np.array([False,True,True]) >>> b2
= np.array([True,False,True,False])
'
>>> a[b1,:]
#lựa chọn mờ đầu tiên
# lựa chọn mờ thứ hai
# chọn hàng
mảng([[ 4,  5,  6,  7],
       [ 8,   9, 10, 11]])
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```

',
>>> a[b1]                                # điều tương tự
mảng([[ 4,  5,  6,  7],
      [ 8,   9, 10, 11]])
',
>>> a[:,b2]                               # chọn cột
mảng([[ 0,  2], [ 4,
                  6], [ 8,
                  10]])
',
>>> a[b1,b2]                            # một điều kỳ lạ để làm
mảng([ 4, 10])

```

Lưu ý rằng độ dài của mảng boolean 1D phải trùng với độ dài của chiều (hoặc trực) bạn muốn cắt.

Trong ví dụ trước, b1 có độ dài 3 (số hàng trong a) và b2 (có độ dài 4) phù hợp để lập chỉ mục cho trực (cột) thứ 2 của a.

### 2.6.3 Hàm ix\_()

Hàm ix\_ có thể được sử dụng để kết hợp các vectơ khác nhau nhằm thu được kết quả cho mỗi n-uplet. Ví dụ: nếu bạn muốn tính tất cả a+b\*c cho tất cả các bộ ba được lấy từ mỗi vectơ a, b và c:

```

>>> a = np.array([2,3,4,5]) >>>
b = np.array([8,5,4]) >>> c =
np.array([5,4, 6,8,3]) >>> ax,bx,cx
= np.ix_(a,b,c)
>>> riu
mảng([[[2],
          [[3]],
          [[4]],
          [[5]]])
>>> bx
mảng([[[8],
          [5],
          [4]]])
>>> cx
array([[[5, 4, 6, 8, 3]]]) >>>
ax.shape, bx.shape, cx.shape ((4, 1, 1),
(1, 3, 1), (1, 1, 5))
>>> kết quả = ax+bx*cx
>>> kết quả
mảng([[[42, 34, 50, 66, 26], [27, 22,
            32, 42, 17],
           [22, 18, 26, 34, 14]],
          [[43, 35, 51, 67, 27],
           [28, 23, 33, 43, 18],
           [23, 19, 27, 35, 15]],
          [[44, 36, 52, 68, 28],
           [29, 24, 34, 44, 19],
           [24, 20, 28, 36, 16]],
          [[45, 37, 53, 69, 29], [30, 25,
            35, 45, 20], [25, 21, 29, 37,
            17]]]) >>> kết quả [3,2,4] 17

```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

&gt;&gt;&gt; a[3]+b[2]\*c[4] 17

Bạn cũng có thể thực hiện việc giảm như sau:

```
>>> def ufunc_reduce(ufct, *vectors): vs =
    np.ix_(*vectors) r =
    ufct.identity cho v
    trong vs:
        r = ufct(r,v) trả
    về r
```

và sau đó sử dụng nó như:

```
>>> ufunc_reduce(np.add,a,b,c)
mảng([[[15, 14, 16, 18, 13], [12, 11,
    13, 15, 10],
    [11, 10, 12, 14, 9]], [[16, 15, 17, 19, 14],
    [13, 12, 14, 16, 11],
    [12, 11, 13, 15, 10]], [[17, 16, 18, 20, 15],
    [14, 13, 15, 17, 12],
    [13, 12, 14, 16, 11]], [[18, 17, 19, 21, 16],
    [15, 14, 16, 18, 13],
    [14, 13, 15, 17, 12]]])
```

Ưu điểm của phiên bản rút gọn này so với `ufunc.reduce` thông thường là nó sử dụng Quy tắc phát sóng . để tránh tạo một mảng đối số có kích thước của đầu ra nhân với số lượng vectơ.

#### 2.6.4 Lập chỉ mục bằng chuỗi

Xem Mảng có cấu trúc.

### 2.7 Đại số tuyến tính

Công việc đang được tiến hành. Đại số tuyến tính cơ bản được đưa vào đây.

#### 2.7.1 Các phép toán mảng đơn giản

Xem `linalg.py` trong thư mục gọn gàng để biết thêm.

```
>>> nhập numpy dưới dạng
np >>> a = np.array([[1.0, 2.0], [3.0, 4.0]]) >>>
print(a) [[ 1.
2.]
 [ 3. 4.]]
>>> a.transpose()
mảng([[ 1., 3.], [ 2.,
4.]])
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
>>> np.linalg.inv(a)
mảng([[-2., 1.], [1.5,
-0.5]])

>>> u = np.eye(2) # ma trận 2x2 đơn vị; "mắt" đại diện cho "tôi"
>>> bạn
mảng([[1., 0.], [0.,
1.]]) >>> j =
np.array([[0.0, -1.0], [1.0, 0.0]])

j mảng([[-1.,
0.], [0.,
-1.]]) # tích ma trận >>> j @

>>> np.trace(u) # dấu vết 2.0

>>> y = np.array([[5.], [7.]]) >>>
np.linalg.solve(a, y) array([-3.,
4.])

>>> np.linalg.eig(j)
(mảng([ 0.+1.j, 0.-1.j]), mảng([[ 0.70710678+0.j
, 0.70710678-0.j
, 0.00000000-0.70710678j, 0.00000000+0.70710678j]]))
```

Thông số:

Ma trận vuông

Trả lại

Các giá trị riêng, mỗi giá trị được lặp lại theo bội số của nó.

Các vectơ riêng được chuẩn hóa (đơn vị "độ dài"), sao cho cột ``v[:,i]`` là vectơ riêng tương ứng với giá trị riêng ``w[i]``.

## 2.8 Thủ thuật và mẹo

Ở đây chúng tôi đưa ra một danh sách các lời khuyên ngắn và hữu ích.

### 2.8.1 Định hình lại “Tự động”

Để thay đổi kích thước của một mảng, bạn có thể bỏ qua một trong các kích thước sau đó sẽ được tự động suy ra:

```
>>> a = np.arange(30) >>>
a.shape = 2,-1,3 # -1 nghĩa là "bất cứ thứ gì cần thiết" >>> a.shape (2, 5, 3)

>>> một
mảng([[[ 0, 1, 2], [ 3, 4, 5],
[ 6, 7, 8], [ 9,
10, 11],
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
[12, 13, 14]],
[[15, 16, 17], [18,
19, 20], [21, 22,
23], [24, 25, 26],
[27, 28, 29]])
```

### 2.8.2 Xếp chồng vectơ

Làm cách nào để xây dựng một mảng 2D từ danh sách các vectơ hàng có kích thước bằng nhau? Trong MATLAB, điều này khá dễ dàng: nếu  $x$  và  $y$  là hai vectơ có cùng độ dài thì bạn chỉ cần thực hiện  $m=[x;y]$ . Trong NumPy, tính năng này hoạt động thông qua các hàm `column_stack`, `dstack`, `hstack` và `vstack`, tùy thuộc vào kích thước mà việc xếp chồng sẽ được thực hiện. Ví dụ:

```
x = np.arange(0,10,2) y =
np.arange(5) m =
np.vstack([x,y])
xy = np.hstack([x,y])
```

Logic đằng sau những chức năng đó trong nhiều chiều có thể rất lạ.

Xem thêm:

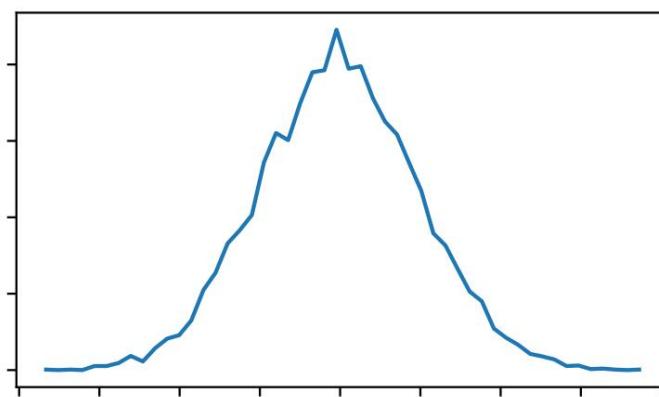
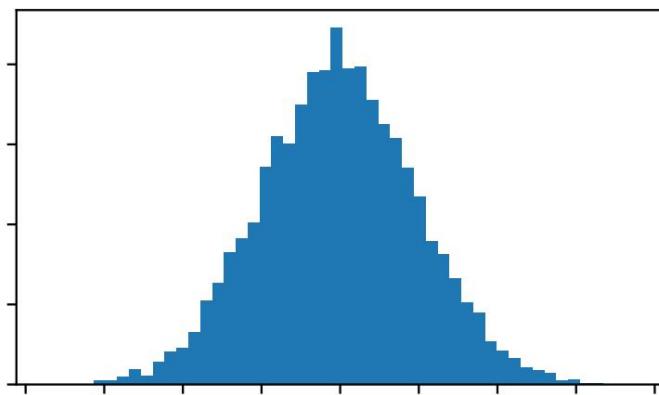
[NumPy cho người dùng Matlab](#)

### 2.8.3 Biểu đồ

Hàm biểu đồ NumPy được áp dụng cho một mảng trả về một cặp vectơ: biểu đồ của mảng và vectơ thùng. Hãy cẩn thận: `matplotlib` cũng có chức năng xây dựng biểu đồ (được gọi là `hist`, như trong Matlab) khác với chức năng trong NumPy. Sự khác biệt chính là `pylab.hist` tự động vẽ biểu đồ, trong khi `numpy.histogram` chỉ tạo dữ liệu.

```
>>> import numpy as np >>> import
matplotlib.pyplot as plt >>> # Xây dựng một vectơ 10000
độ lệch chuẩn với phương sai 0,5^2 và trung bình 2
>>> mu, sigma = 2, 0,5 >>> v =
np.random.normal(mu, sigma, 10000)
>>> # Vẽ biểu đồ chuẩn hóa với 50 bin >>> plt.hist(v, bins=50, Density=1)
>>> plt.show() # phiên bản matplotlib (cốt truyện)
```

```
>>> # Tính biểu đồ với numpy rồi vẽ đồ thị >>> (n, bins) = np.histogram(v, bins=50,
Density=True) # Phiên bản NumPy (không có cốt truyện) >>> plt.plot( .5*(bins[1:]+bins[:-1]), n) >>> plt.show()
```



## 2.9 Đọc thêm

- Hướng dẫn Python
  - người giới thiệu
- Hướng dẫn SciPy
- Ghi chú bài giảng SciPy
- Từ điển matlab, R, IDL, NumPy/SciPy

## 3.1 Kiểu dữ liệu

Xem thêm:

[đổi tương kiểu dữ liệu](#)

### 3.1.1 Kiểu mảng và chuyển đổi giữa các kiểu

NumPy hỗ trợ nhiều loại số hơn Python. Phần này hiển thị những gì có sẵn, và cách sửa đổi kiểu dữ liệu của mảng.

Các kiểu nguyên thủy được hỗ trợ được gán chung với các kiểu trong C:

Numpy loại C loại	Sự miêu tả
<code>np.bool_</code>	<code>bool</code> Boolean (Đúng hoặc Sai) được lưu trữ dưới dạng byte
<code>np.byte</code>	<code>đã ký char</code> Nền tảng xác định
<code>np.ubyte</code>	<code>unsigned char</code> Nền tảng được xác định
<code>np.short</code>	<code>ngắn</code> Nền tảng xác định
<code>np.ushort</code>	<code>unsigned short</code> Được xác định theo nền tảng
<code>np.intc</code>	<code>int</code> Nền tảng xác định
<code>np.uintc</code>	<code>unsigned int np.int_long</code> Nền tảng xác định
<code>np.uint</code>	<code>unsigned long</code> Nền tảng xác định
<code>Nền tảng xác định</code>	
<code>np.longlong</code>	Được xác định dài
<code>np.ulonglong</code>	không dấu dài
<code>np.half</code>	Phao chính xác một nửa: bit dấu, số mũ 5 bit, lớp định trị 10 bit
<code>np.float16</code>	
<code>np.single</code>	trôi nổi Phao chính xác đơn do nền tảng xác định: thường là bit ký, số mũ 8 bit, phần định trị 23 bit
<code>np.double</code>	gấp đôi Phao chính xác kép do nền tảng xác định: thường là bit ký, số mũ 11 bit, phần định trị 52 bit.
<code>np.longdouble</code>	long double xác mở rộng được xác định trên nền tảng
<code>np.cssingle</code>	float phức tạp Số phức, được biểu thị bằng hai số float có độ chính xác đơn (thực và phần ảo)
<code>np.cdouble</code>	Số phức phức kép, được biểu thị bằng hai số float có độ chính xác kép (thực và phần ảo).
<code>np.clongdouble</code>	dài gấp đôi tổ hợp Số phức, được biểu thị bằng hai số float có độ chính xác mở rộng (thực và phần ảo).

Vì nhiều trong số này có định nghĩa phụ thuộc vào nền tảng nên một tập hợp các bí danh có kích thước cố định được cung cấp:

Kiểu NumPy	Kiểu C	Sự miêu tả
np.int8	int8_t	Byte (-128 đến 127)
np.int16	int16_t	Số nguyên (-32768 đến 32767)
np.int32	int32_t	Số nguyên (-2147483648 đến 2147483647)
np.int64	int64_t	Số nguyên (-9223372036854775808 đến 9223372036854775807)
np.uint8	uint8_t	Số nguyên không dấu (0 đến 255)
np.uint16	uint16_t	Số nguyên không dấu (0 đến 65535)
np.uint32	uint32_t	Số nguyên không dấu (0 đến 4294967295)
np.uint64	uint64_t	Số nguyên không dấu (0 đến 18446744073709551615)
np.intp	intptr_t	Số nguyên dùng để lập chỉ mục, thường giống như ssize_t.
np.uintp	uintptr_t	Số nguyên đủ lớn để chứa một con trỏ
np.float32	trôi nổi	
np.float64 / np.float_	gấp đôi	Lưu ý rằng điều này phù hợp với độ chính xác của float python dựng sẵn.
np.complex64	trôi nổi tổ hợp	Số phức, được biểu thị bằng hai số float 32 bit (thành phần thực và ảo)
np.complex128	, gấp đôi	Lưu ý rằng điều này phù hợp với độ chính xác của phức hợp python dựng sẵn.
np.complex_	tổ hợp	

Các kiểu số NumPy là các thể hiện của đối tượng dtype (kiểu dữ liệu), mỗi đối tượng có các đặc điểm riêng. Một khi bạn đã nhập NumPy bằng cách sử dụng

```
>>> nhập numpy dưới dạng np
```

các dtype có sẵn dưới dạng np.bool\_, np.float32, v.v.

Các loại nâng cao, không được liệt kê trong bảng trên, được khám phá trong phần [Mảng có cấu trúc](#).

Có 5 kiểu số cơ bản biểu diễn booleans (bool), số nguyên (int), số nguyên không dấu (uint) dấu phẩy động (phao) và phức tạp. Những cái có số trong tên cho biết kích thước bit của loại (tức là cần bao nhiêu bit để biểu thị một giá trị duy nhất trong bộ nhớ). Một số loại, chẳng hạn như int và intp, có kích thước bit khác nhau, phụ thuộc vào nền tảng (ví dụ: máy 32 bit so với máy 64 bit). Điều này cần được tính đến khi giao tiếp với mã cấp thấp (chẳng hạn như C hoặc Fortran) nơi bộ nhớ thô được xử lý.

Kiểu dữ liệu có thể được sử dụng làm hàm để chuyển đổi số python thành vô hướng mảng (xem phần [vô hướng mảng để biết giải thích](#)), chuỗi số python cho các mảng thuộc loại đó hoặc làm đối số cho từ khóa dtype mà nhiều các hàm hoặc phương thức gọn gàng được chấp nhận. Vài ví dụ:

```
>>> nhập numpy dưới dạng np
>>> x = np.float32(1.0)
>>> x
1.0
>>> y = np.int_([1,2,4])
>>> y
mảng([1, 2, 4])
>>> z = np.arange(3, dtype=np.uint8)
>>> z
mảng([0, 1, 2], dtype=uint8)
```

Các loại mảng cũng có thể được gọi bằng mã ký tự, chủ yếu là để duy trì khả năng tương thích ngược với các gói cũ hơn. chẳng hạn như số. Một số tài liệu vẫn có thể đề cập đến những điều này, ví dụ:

```
>>> np.array([1, 2, 3], dtype='f')
mảng([ 1., 2., 3.], dtype=float32)
```

Thay vào đó, chúng tôi khuyên bạn nên sử dụng các đối tượng dtype.

Để chuyển đổi kiểu của một mảng, hãy sử dụng phương thức `.astype()` (ưu tiên) hoặc chính kiểu đó làm hàm. Ví dụ:

```
>>> z.astype(float)
array([ 0., 1., 2.]) >>>
np.int8(z)
array([0, 1, 2], dtype=int8)
```

Lưu ý rằng ở trên, chúng tôi sử dụng đối tượng float của Python làm dtype. NumPy biết rằng int để cập đến np.int\_, bool có nghĩa là np.bool\_, float đó là np.float\_ và complex là np.complex\_. Các kiểu dữ liệu khác không có kiểu dữ liệu tương đương với Python.

Để xác định loại mảng, hãy xem thuộc tính dtype:

```
>>> z.dtype
```

Các đối tượng dtype cũng chứa thông tin về loại, chẳng hạn như độ rộng bit và thứ tự byte của nó. Kiểu dữ liệu cũng có thể được sử dụng gián tiếp để truy vấn các thuộc tính của loại, chẳng hạn như liệu nó có phải là số nguyên hay không:

```
>>> d = np.dtype(int) >>>
d
dtype('int32')

>>> np.issubdtype(d, np.integer)
DÙNG VÀY

>>> np.issubdtype(d, np.floating)
SAI
```

### 3.1.2 Mảng vô hướng

NumPy thường trả về các phần tử của mảng dưới dạng mảng vô hướng (một đại lượng vô hướng có dtype liên quan). Đại lượng vô hướng mảng khác với đại lượng vô hướng Python, nhưng trong hầu hết các trường hợp, chúng có thể được sử dụng thay thế cho nhau (ngoại lệ chính là dành cho các phiên bản Python cũ hơn v2.x, trong đó đại số vô hướng mảng số nguyên không thể đóng vai trò là chỉ mục cho danh sách và bộ dữ liệu). Có một số trường hợp ngoại lệ, chẳng hạn như khi mã yêu cầu các thuộc tính rất cụ thể của đại lượng vô hướng hoặc khi mã kiểm tra cụ thể xem một giá trị có phải là đại lượng vô hướng Python hay không. Nói chung, các vấn đề có thể dễ dàng khắc phục bằng cách chuyển đổi rõ ràng các giá trị vô hướng của mảng thành các giá trị vô hướng Python, sử dụng hàm kiểu Python tương ứng (ví dụ: int, float, complex, str, unicode).

Ưu điểm chính của việc sử dụng mảng là chúng bảo toàn kiểu mảng (Python có thể không có sẵn kiểu vô hướng phù hợp, ví dụ: int16). Do đó, việc sử dụng mảng vô hướng đảm bảo hành vi giống hệt nhau giữa mảng và vô hướng, bất kể giá trị có nằm trong mảng hay không. Các hàm vô hướng NumPy cũng có nhiều phương thức tương tự như mảng.

### 3.1.3 Lỗi tràn

Kích thước cố định của kiểu số NumPy có thể gây ra lỗi tràn khi một giá trị yêu cầu nhiều bộ nhớ hơn mức có sẵn trong kiểu dữ liệu. Ví dụ: `numpy.power` đánh giá chính xác  $100 * 10^{** 8}$  cho số nguyên 64 bit, nhưng cho kết quả 1874919424 (không chính xác) đối với số nguyên 32 bit.

```
>>> np.power(100, 8, dtype=np.int64)
1000000000000000000000000
>>> np.power(100, 8, dtype=np.int32) 1874919424
```

Hành vi của các loại số nguyên NumPy và Python khác nhau đáng kể khi tròn số nguyên và có thể khiến người dùng nhầm lẫn mong đợi các số nguyên NumPy hoạt động tương tự như int của Python. Không giống như NumPy, kích thước int của Python rất linh hoạt. Điều này có nghĩa là số nguyên Python có thể mở rộng để chứa bất kỳ số nguyên nào và sẽ không bị tròn.

NumPy cung cấp numpy.iinfo và numpy.finfo để xác minh giá trị tối thiểu hoặc tối đa của số nguyên NumPy và giá trị dấu phẩy động tương ứng.

```
>>> np.iinfo(int) # Giới hạn của số nguyên mặc định trên hệ thống này. iinfo(min=-9223372036854775808,
max=9223372036854775807, dtype=int64) >>> np.iinfo(np.int32) # Giới hạn của số nguyên 32 bit
iinfo(min=-2147483648, max=2147483647, dtype=int32) >>> np.iinfo(np.int64)
# Giới hạn của số nguyên 64 bit iinfo(min=-9223372036854775808,
max=9223372036854775807, dtype=int64)
```

Nếu số nguyên 64 bit vẫn còn quá nhỏ thì kết quả có thể được chuyển thành số dấu phẩy động. Số dấu phẩy động cung cấp phạm vi giá trị có thể lớn hơn nhưng không chính xác.

```
>>> np.power(100, 100, dtype=np.int64) # Không đúng ngay cả với 64-bit int 0
>>> np.power(100, 100, dtype=np.float64) 1e+200
```

### 3.1.4 Độ chính xác mở rộng

Số dấu phẩy động của Python thường là số dấu phẩy động 64 bit, gần tương đương với np.float64. Trong một số trường hợp bất thường, việc sử dụng số dấu phẩy động với độ chính xác cao hơn có thể hữu ích. Việc điều này có khả thi hay không phụ thuộc vào phần cứng và môi trường phát triển: cụ thể, máy x86 cung cấp dấu phẩy động phần cứng với độ chính xác 80 bit và trong khi hầu hết các trình biên dịch C cung cấp đây là loại kép dài, MSVC (tiêu chuẩn cho các bản dựng Windows) làm cho long double giống hệt double (64 bit). NumPy cung cấp long double của trình biên dịch dưới dạng np.longdouble (và np.clongdouble cho các số phức). Bạn có thể tìm hiểu xem Numpy của bạn cung cấp những gì với np.finfo(np.longdouble).

NumPy không cung cấp dtype có độ chính xác cao hơn double dài của C; đặc biệt, loại dữ liệu chính xác 4 góc IEEE 128-bit (FORTRAN's REAL\*16) không có sẵn.

Để căn chỉnh bộ nhớ hiệu quả, np.longdouble thường được lưu trữ với các bit 0, đến 96 hoặc 128 bit. Cái nào hiệu quả hơn phụ thuộc vào phần cứng và môi trường phát triển; thông thường trên hệ thống 32 bit, chúng được đệm thành 96 bit, trong khi trên hệ thống 64 bit, chúng thường được đệm thành 128 bit. np.longdouble được đệm theo mặc định của hệ thống; np.float96 và np.float128 được cung cấp cho những người dùng muốn có phần đệm cụ thể. Bất chấp tên gọi, np.float96 và np.float128 chỉ cung cấp độ chính xác tương đương với np.longdouble, tức là 80 bit trên hầu hết các máy x86 và 64 bit trong các bản dựng Windows tiêu chuẩn.

Xin lưu ý rằng ngay cả khi np.longdouble cung cấp độ chính xác cao hơn python float, bạn vẫn có thể dễ dàng mất đi độ chính xác cao hơn đó, vì python thường buộc các giá trị đi qua float. Ví dụ: toán tử định dạng % yêu cầu các đối số của nó phải được chuyển đổi thành các loại python tiêu chuẩn và do đó không thể duy trì độ chính xác mở rộng ngay cả khi yêu cầu nhiều vị trí thập phân. Việc kiểm tra mã của bạn với giá trị  $1 + np.finfo(np.longdouble).eps$  có thể hữu ích.

## 3.2 Tạo mảng

Xem thêm:

[Thủ tục tạo mảng](#)

### 3.2.1 Giới thiệu

Có 5 cơ chế chung để tạo mảng:

- 1) Chuyển đổi từ các cấu trúc Python khác (ví dụ: danh sách, bộ dữ liệu)
- 2) Các đối tượng tạo mảng có nhiều mảng nội tại (ví dụ: `array`, số một, số không, v.v.)
- 3) Đọc mảng từ đĩa, từ định dạng tiêu chuẩn hoặc tùy chỉnh
- 4) Tạo mảng từ byte thông qua việc sử dụng chuỗi hoặc bộ đệm
- 5) Sử dụng các chức năng thư viện đặc biệt (ví dụ: `ngẫu nhiên`)

Phần này sẽ không bao gồm các phương tiện sao chép, nối hoặc mở rộng hoặc thay đổi các mảng hiện có. Nó cũng sẽ không bao gồm việc tạo mảng đối tượng hoặc mảng có cấu trúc. Cả hai đều được đề cập trong phần riêng của họ.

### 3.2.2 Chuyển đổi các đối tượng giống mảng Python thành mảng NumPy

Nói chung, dữ liệu số được sắp xếp theo cấu trúc giống mảng trong Python có thể được chuyển đổi thành mảng thông qua việc sử dụng hàm `array()`. Các ví dụ rõ ràng nhất là danh sách và bộ dữ liệu. Xem tài liệu về `array()` để biết chi tiết về cách sử dụng nó. Một số đối tượng có thể hỗ trợ giao thức mảng và cho phép chuyển đổi thành mảng theo cách này. Một cách đơn giản để tìm hiểu xem đối tượng có thể được chuyển đổi thành một mảng gọn gàng bằng cách sử dụng `array()` hay không chỉ đơn giản là thử tương tác và xem nó có hoạt động không! (Con đường Python).

Ví dụ:

```
>>> x = np.array([2,3,1,0]) >>> x =
np.array([2, 3, 1, 0]) >>> ,2.0],[0,0] ,
(1+1j,3.)) # note kết hợp giữa bộ và danh sách,
và các loại
>>> - - - - - - - - - 3.+0.j]])
```

### 3.2.3 Tạo mảng NumPy nội tại

NumPy có các hàm dựng sẵn để tạo mảng từ đầu:

`zeros(shape)` sẽ tạo một mảng chứa đầy các giá trị 0 với hình dạng được chỉ định. Dtype mặc định là `float64`.

```
>>> np.zeros((2, 3))
array([[ 0.,  0.,  0.], [ 0.,  0.,  0.]])
```

những cái(hình dạng) sẽ tạo ra một mảng chứa 1 giá trị. Nó giống hệt số không ở tất cả các khía cạnh khác.

`sắp xếp()` sẽ tạo các mảng có giá trị tăng dần đều đặn. Kiểm tra chuỗi tài liệu để biết thông tin đầy đủ về các cách khác nhau có thể sử dụng chuỗi đó. Một vài ví dụ sẽ được đưa ra ở đây:

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]) >>>
np.arange(2, 10, dtype=float) array([ 2.,
3., 4., 5., 6., 7., 8., 9.]) >>> np.arange(2, 3,
0.1) array([ 2., 2.1, 2.2,
2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])
```

Lưu ý rằng có một số điểm tinh tế liên quan đến lần sử dụng cuối cùng mà người dùng nên biết được mô tả trong chuỗi tài liệu arange.

linspace() sẽ tạo các mảng có số phần tử được chỉ định và cách đều nhau giữa các giá trị bắt đầu và kết thúc được chỉ định. Ví dụ:

```
>>> np.linspace(1., 4., 6)
array([ 1., 1.6, 2.2, 2.8, 3.4, 4.])
```

Ưu điểm của hàm tạo này là người ta có thể đảm bảo số lượng phần tử cũng như điểm bắt đầu và điểm kết thúc, điều này sắp xếp() thường sẽ không thực hiện đổi với các giá trị bắt đầu, dừng và bước tùy ý.

index() sẽ tạo một tập hợp các mảng (được xếp chồng lên nhau dưới dạng mảng một chiều cao hơn), mỗi mảng cho mỗi thứ nguyên với mỗi biến thể ghi đại diện trong thứ nguyên đó. Một ví dụ minh họa tốt hơn nhiều so với mô tả bằng lời nói:

```
>>> np.indices((3,3))
array([[[0, 0, 0], [1, 1, 1], [2, 2, 2]], [[0, 1, 2], [0, 1, 2], [0, 1, 2]]])
```

Điều này đặc biệt hữu ích để đánh giá các hàm có nhiều chiều trên lưới thông thường.

### 3.2.4 Đọc mảng từ đĩa

Đây có lẽ là trường hợp phổ biến nhất của việc tạo mảng lớn. Tuy nhiên, các chi tiết phụ thuộc rất nhiều vào định dạng dữ liệu trên đĩa và do đó phần này chỉ có thể đưa ra những gợi ý chung về cách xử lý các định dạng khác nhau.

#### Định dạng nhị phân tiêu chuẩn

Các trường khác nhau có định dạng chuẩn cho dữ liệu mảng. Phần sau đây liệt kê những cái có thư viện python đã biết để đọc chúng và trả về các mảng có nhiều mảng (có thể có những mảng khác có thể đọc và chuyển đổi thành mảng có nhiều mảng, vì vậy hãy kiểm tra phần cuối cùng)

<b>hdf5:h5py</b>
PHÙ HỢP VỚI: Thiên văn học

Ví dụ về các định dạng không thể đọc trực tiếp nhưng không khó chuyển đổi là những định dạng được hỗ trợ bởi các thư viện như PIL (có thể đọc và ghi nhiều định dạng hình ảnh như jpg, png, v.v.).

#### Các định dạng ASCII phổ biến

Các tệp Giá trị được phân tách bằng dấu phẩy (CSV) được sử dụng rộng rãi (và tùy chọn xuất và nhập cho các chương trình như Excel). Có một số cách để đọc các tệp này bằng Python. Có các hàm CSV trong Python và các hàm trong pylab (một phần của matplotlib).

Có thể đọc các tệp ascii chung hơn bằng cách sử dụng gói io trong scipy.

## Định dạng nhị phân tùy chỉnh

Có nhiều cách tiếp cận khác nhau mà người ta có thể sử dụng. Nếu tệp có định dạng tương đối đơn giản thì người ta có thể viết một thư viện I/O đơn giản và sử dụng hàm `fromfile()` gọn gàng và phương thức `.tofile()` để đọc và ghi trực tiếp các mảng có nhiều nếp nhăn (mặc dù vậy, hãy chú ý đến thứ tự byte của bạn!). Nếu một C tốt hoặc thư viện C++ tồn tại để đọc dữ liệu, người ta có thể gói thư viện đó bằng nhiều kỹ thuật khác nhau mặc dù điều đó chắc chắn tồn nhiều công sức hơn và đòi hỏi kiến thức nâng cao hơn đáng kể để giao tiếp với C hoặc C++.

## Sử dụng thư viện đặc biệt

Có những thư viện có thể được sử dụng để tạo mảng cho các mục đích đặc biệt và không thể liệt kê tất cả chúng. Cách sử dụng phổ biến nhất là sử dụng nhiều hàm tạo mảng một cách ngẫu nhiên có thể tạo ra các mảng giá trị ngẫu nhiên và một số hàm tiện ích để tạo ra các ma trận đặc biệt (ví dụ: đường chéo).

## 3.3 I/O với NumPy

### 3.3.1 Nhập dữ liệu bằng `genfromtxt`

NumPy cung cấp một số hàm để tạo mảng từ dữ liệu dạng bảng. Ở đây chúng tôi tập trung vào hàm `genfromtxt`.

Tóm lại, `genfromtxt` chạy hai vòng lặp chính. Vòng lặp đầu tiên chuyển đổi từng dòng của tệp thành một chuỗi các chuỗi. Vòng lặp thứ hai chuyển đổi từng chuỗi thành kiểu dữ liệu thích hợp. Cơ chế này chậm hơn so với cơ chế lặp đơn nhưng mang lại sự linh hoạt hơn. Đặc biệt, `genfromtxt` có thể tính đến dữ liệu bị thiếu trong khi các chức năng nhanh hơn và đơn giản hơn như `Loadtxt` không thể.

---

Lưu ý: Khi đưa ra ví dụ, chúng ta sẽ sử dụng các quy ước sau:

```
>>> nhập numpy dưới dạng
np >>> từ io import StringIO
```

---

#### Xác định đầu vào

Đối số bắt buộc duy nhất của `genfromtxt` là nguồn dữ liệu. Nó có thể là một chuỗi, một danh sách các chuỗi, một trình tạo hoặc một đối tượng giống như tệp đang mở với phương thức đọc, ví dụ: tệp hoặc `io.StringIO` sự vật. Nếu một chuỗi được cung cấp, nó được coi là tên của tệp cục bộ hoặc từ xa. Nếu danh sách các chuỗi hoặc chuỗi trả về được cung cấp thì mỗi chuỗi được coi là một dòng trong một tệp. Khi URL của tệp từ xa được chuyển, tệp sẽ tự động được tải xuống thư mục hiện tại và được mở.

Các loại tệp được công nhận là tệp văn bản và tệp lưu trữ. Hiện tại, chức năng này nhận dạng các kho lưu trữ gzip và bz2 (bzip2). Loại tệp lưu trữ được xác định từ phần mở rộng của tệp: nếu tên tệp kết thúc bằng '.gz', thì tệp lưu trữ gzip sẽ được mong đợi; nếu nó kết thúc bằng 'bz2', thì kho lưu trữ bzip2 được giả định.

### Chia dòng thành cột

#### Đối số dấu phân cách

Sau khi tệp được xác định và mở để đọc, genfromtxt sẽ chia từng dòng không trống thành một chuỗi chuỗi.

Các dòng trống hoặc nhận xét sẽ bị bỏ qua. Từ khóa dấu phân cách được sử dụng để xác định cách thức phân chia.

địa điểm.

Thông thường, một ký tự đơn đánh dấu sự phân cách giữa các cột. Ví dụ: các tệp được phân tách bằng dấu phẩy (CSV) sử dụng dấu phẩy (,) hoặc dấu chấm phẩy (;) làm dấu phân cách:

```
>>> dữ liệu = u"1, 2, 3\n4, 5, 6"
>>> np.genfromtxt(StringIO(data), delimiter=",")
mảng([[1., 2., 3.],
       [4., 5., 6.]])
```

Một dấu phân cách phổ biến khác là "\t", ký tự lập bảng. Tuy nhiên, chúng tôi không bị giới hạn ở một ký tự duy nhất, bất kỳ chuỗi sẽ làm. Theo mặc định, genfromtxt giả sử dấu phân cách = Không, nghĩa là dòng được phân chia đọc theo màu trắng khoảng trắng (bao gồm cả tab) và các khoảng trắng liên tiếp được coi là một khoảng trắng duy nhất.

Ngoài ra, chúng ta có thể xử lý một tệp có chiều rộng cố định, trong đó các cột được xác định bằng một số ký tự nhất định.

Trong trường hợp đó, chúng ta cần đặt dấu phân cách thành một số nguyên duy nhất (nếu tất cả các cột có cùng kích thước) hoặc thành một chuỗi số nguyên (nếu các cột có thể có kích thước khác nhau):

```
>>> dữ liệu = u" 1 2 3\n 4 5 67\n890123 4"
>>> np.genfromtxt(StringIO(dữ liệu), dấu phân cách=3)
mảng([[1., 2., 3.],
       [4., 5., 67.],
       [890., 123., 4.]])
>>> dữ liệu = u"123456789\n 4 7 9\n 4567 9"
>>> np.genfromtxt(StringIO(data), delimiter=(4, 3, 2))
mảng([[1234., 567., 89.],
       [4., 7., 9.],
       [4., 567., 9.]])
```

#### Đối số dài tự động

Theo mặc định, khi một dòng được phân tách thành một chuỗi các chuỗi, các mục riêng lẻ sẽ không bị loại bỏ phần đầu cũng như phần đầu các khoảng trắng ở cuối. Hành vi này có thể được ghi đè bằng cách đặt dài tự động đối số tùy chọn thành giá trị ĐÚNG VẬY:

```
>>> data = u"1, abc >>> , 2\n 3, xxx, 4"
# Không có dài tự động
>>> np.genfromtxt(StringIO(data), delimiter=",", dtype="|U5")
mảng([['1', 'abc', '2'],
      ['3', 'xxx', '4']], dtype='<U5')
>>> # Với dài tự động
>>> np.genfromtxt(StringIO(data), delimiter=",", dtype="|U5", autostrip=True)
mảng([['1', 'abc', '2'],
      ['3', 'xxx', '4']], dtype='<U5')
```

### Đối số nhận xét

Đối số tùy chọn comment được sử dụng để xác định một chuỗi ký tự đánh dấu sự bắt đầu của một comment. Theo mặc định, genfromtxt giả định comments='#'. Điểm đánh dấu nhận xét có thể xuất hiện ở bất kỳ đâu trên dòng. Bất kỳ ký tự nào xuất hiện sau (các) dấu nhận xét đều bị bỏ qua:

```
>>> data = u"""# ...
# Bỏ qua tôi !
... # Bỏ qua tôi
nữa ! ... 1, 2
... 3,
4 ... 5, 6 #Đây là dòng dữ liệu thứ 3
... 7, 8 ...
# Và đây là dòng cuối cùng ... 9, 0

,
>>> np.genfromtxt(StringIO(data), comments="#", delimiter=",") array([[1.,
2.], [3., 4.], [5.,
6.], [7.,
8.], [9.,
0.]])
```

Tính năng mới trong phiên bản 1.7.0: Khi nhận xét được đặt thành Không có, không có dòng nào được coi là nhận xét.

---

Lưu ý: Có một ngoại lệ đáng chú ý đối với hành vi này: nếu đối số tùy chọn tên=True, dòng nhận xét đầu tiên sẽ được kiểm tra tên.

---

### Bỏ qua dòng và chọn cột

#### Các đối số Skip\_header và Skip\_footer

Sự hiện diện của tiêu đề trong tệp có thể cản trở việc xử lý dữ liệu. Trong trường hợp đó, chúng ta cần sử dụng đối số tùy chọn Skip\_header. Các giá trị của đối số này phải là số nguyên tương ứng với số dòng cần bỏ qua ở đầu tệp, trước khi thực hiện bất kỳ hành động nào khác. Tương tự, chúng ta có thể bỏ qua n dòng cuối cùng của tệp bằng cách sử dụng thuộc tính Skip\_footer và gán cho nó giá trị n:

```
>>> data = u"\n".join(str(i) for i in range(10)) >>>
np.genfromtxt(StringIO(data),
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.]) >>>
np.genfromtxt(StringIO(data),
,
Skip_header=3, Skip_footer=5)
mảng([ 3., 4.])
```

Theo mặc định, Skip\_header=0 và Skip\_footer=0, nghĩa là không có dòng nào bị bỏ qua.

### Đối số usecols

Trong một số trường hợp, chúng ta không quan tâm đến tất cả các cột của dữ liệu mà chỉ quan tâm đến một vài cột trong số đó. Chúng ta có thể chọn cột nào sẽ nhập bằng đối số usecols. Đối số này chấp nhận một số nguyên hoặc một chuỗi số nguyên tương ứng với chỉ mục của các cột cần nhập. Hãy nhớ rằng theo quy ước, cột đầu tiên có chỉ mục là 0. Số nguyên âm hoạt động giống như chỉ mục Python thông thường.

Ví dụ: nếu chỉ muốn nhập cột đầu tiên và cột cuối cùng, chúng ta có thể sử dụng usecols=(0, -1):

```
>>> data = u"1 2 3\n4 5 6" >>>
np.genfromtxt(StringIO(data), usecols=(0, -1)) array([[ 1.,
3.], [ 4., 6.]])
```

Nếu các cột có tên, chúng ta cũng có thể chọn cột nào sẽ nhập bằng cách đặt tên của chúng cho đối số usecols, dưới dạng chuỗi chuỗi hoặc chuỗi được phân tách bằng dấu phẩy:

```
>>> data = u"1 2 3\n4 5 6" >>>
np.genfromtxt(StringIO(data),
              name="a, b, c", usecols=("a", "c"))
array([(1.0, 3.0), (4.0, 6.0)],
      dtype=[('a', '<f8'), ('c', '<f8')])
>>> np.genfromtxt(StringIO(dữ liệu),
              name="a, b, c", usecols=("a, c"))
array([(1.0, 3.0), (4.0, 6.0)],
      dtype=[('a', '<f8'), ('c', '<f8')])
```

### Lựa chọn kiểu dữ liệu

Cách chính để kiểm soát cách chuyển đổi các chuỗi chuỗi chúng ta đã đọc từ tệp sang các loại khác là đặt đối số dtype. Các giá trị được chấp nhận cho đối số này là:

- một kiểu duy nhất, chẳng hạn như dtype=float. Đầu ra sẽ là 2D với dtype đã cho, trừ khi tên được liên kết với mỗi cột bằng cách sử dụng đối số tên (xem bên dưới). Lưu ý rằng dtype=float là mặc định cho genfromtxt.
- một chuỗi các kiểu, chẳng hạn như dtype=(int, float, float).
- một chuỗi được phân tách bằng dấu phẩy, chẳng hạn như dtype="i4,f8,|U3".
- một từ điển có hai khóa 'tên' và 'định dạng'.
- một chuỗi các bộ (tên, loại), chẳng hạn như dtype=[('A', int), ('B', float)].
- một đối tượng numpy.dtype hiện có.
- giá trị đặc biệt Không có. Trong trường hợp đó, loại cột sẽ được xác định từ chính dữ liệu đó (xem bên dưới).

Trong tất cả các trường hợp ngoại trừ trường hợp đầu tiên, đầu ra sẽ là mảng 1D với kiểu dữ liệu có cấu trúc. Dtype này có nhiều trường như các mục trong chuỗi. Tên trường được xác định bằng từ khóa tên.

Khi dtype=None, loại của mỗi cột được xác định lặp lại từ dữ liệu của nó. Chúng tôi bắt đầu bằng cách kiểm tra xem một chuỗi có thể được chuyển đổi thành boolean hay không (nghĩa là liệu chuỗi đó có khớp đúng hay sai trong các chữ thường); sau đó liệu nó có thể được chuyển đổi thành số nguyên, sau đó thành số float, sau đó thành số phức và cuối cùng thành chuỗi hay không. Hành vi này có thể được thay đổi bằng cách sửa đổi trình ánh xạ mặc định của lớp StringConverter.

Tùy chọn dtype=None được cung cấp để thuận tiện. Tuy nhiên, nó chậm hơn đáng kể so với việc thiết lập dtype một cách rõ ràng.

## Đặt tên

### Đối số tên

Một cách tiếp cận tự nhiên khi xử lý dữ liệu dạng bảng là đặt tên cho mỗi cột. Khả năng đầu tiên là sử dụng một dtype có cấu trúc rõ ràng, như đã đề cập trước đó:

```
>>> data = StringIO("1 2 3\n4 5 6") >>>
np.loadtxt(data, dtype=[('a', int) cho mảng([(1, 2, 3), . . . trong "abc"])
(4, 5, 6)],
dtype=[('a', '<i8'), ('b', '<i8'), ('c', '<i8')])
```

Một khả năng đơn giản hơn là sử dụng từ khóa tên với một chuỗi các chuỗi hoặc chuỗi được phân tách bằng dấu phẩy:

```
>>> data = StringIO("1 2 3\n4 5 6") >>>
np.loadtxt(data, name="A, B, C") array([(1.0, 2.0,
3.0), (4.0, 5.0, 6.0)], dtype=[('A', '<f8'), ('B',
'<f8'), ('C', '<f8')])
```

Trong ví dụ trên, chúng tôi đã sử dụng thực tế là theo mặc định, dtype=float. Bằng cách đưa ra một chuỗi tên, chúng tôi buộc đầu ra phải có một loại dtype có cấu trúc.

Đôi khi chúng ta có thể cần xác định tên cột từ chính dữ liệu đó. Trong trường hợp đó, chúng ta phải sử dụng từ khóa tên có giá trị True. Sau đó, tên sẽ được đọc từ dòng đầu tiên (sau dòng Skip\_header), ngay cả khi dòng đó được nhận xét:

```
>>> data = StringIO("Vậy là xong \n#a b c\n1 2 3\n4 5 6") >>>
np.loadtxt(data, skip_header=1, name=True) array([(1.0,
2.0, 3.0), (4.0, 5.0, 6.0)], dtype=[('a',
'<f8'), ('b', '<f8'), ('c', '<f8')])
```

Giá trị mặc định của tên là Không có. Nếu chúng tôi cung cấp bất kỳ giá trị nào khác cho từ khóa, tên mới sẽ ghi đè tên trường mà chúng tôi có thể đã xác định bằng dtype:

```
>>> data = StringIO("1 2 3\n4 5 6") >>>
ndtype=[('a',int), ('b', float), ('c', int)] >>> name
= ["A", "B", "C"] >>>
np.loadtxt(data, name=names, dtype=ndtype) array([(1,
2.0, 3), (4, 5.0, 6)],
dtype=[('A', '<i8'), ('B', '<f8'), ('C', '<i8')])
```

### Đối số defaultfmt

Nếu tên=None nhưng dự kiến sẽ có một loại dtype có cấu trúc, thì tên sẽ được xác định với mặc định NumPy tiêu chuẩn là "%fi", mang lại các tên như f0, f1, v.v.:

```
>>> data = StringIO("1 2 3\n4 5 6") >>>
np.loadtxt(data, dtype=(int, float, int)) array([(1,
2.0, 3), (4, 5.0, 6)],
dtype=[('f0', '<i8'), ('f1', '<f8'), ('f2', '<i8')])
```

Theo cách tương tự, nếu chúng ta không cung cấp đủ tên để khớp với độ dài của dtype, thì những tên bị thiếu sẽ được xác định bằng mẫu mặc định này:

```
>>> data = StringIO("1 2 3\n4 5 6") >>>
np.genfromtxt(data, dtype=(int, float, int), name="a") array([(1, 2.0, 3), (4, 5.0,
6)], dtype=[('a', '<i8'), ('f0', '<f8'), ('f1', '<i8')])
```

Chúng ta có thể ghi đè mặc định này bằng đổi số `defaultfmt`, đổi số này nhận bất kỳ chuỗi định dạng nào:

```
>>> data = StringIO("1 2 3\n4 5 6") >>>
np.genfromtxt(data, dtype=(int, float, int), defaultfmt="var_%02i") array([(1, 2.0, 3), (4, 5.0, 6)],

dtype=[('var_00', '<i8'), ('var_01', '<f8'), ('var_02', '<i8')])
```

Lưu ý: Chúng ta cần lưu ý rằng `defaultfmt` chỉ được sử dụng nếu một số tên được mong đợi nhưng chưa được xác định.

#### Xác thực tên

Mảng NumPy có dtype có cấu trúc cũng có thể được xem dưới dạng `recarray`, trong đó một trường có thể được truy cập như thể nó là một thuộc tính. Vì lý do đó, chúng tôi có thể cần đảm bảo rằng tên trường không chứa bất kỳ khoảng trắng hoặc ký tự không hợp lệ nào hoặc nó không tương ứng với tên của thuộc tính tiêu chuẩn (như kích thước hoặc hình dạng), điều này sẽ gây nhầm lẫn cho trình thông dịch. `genfromtxt` chấp nhận ba đổi số tùy chọn cung cấp khả năng kiểm soát tên tốt hơn:

`deletechars` Cung cấp một chuỗi kết hợp tất cả các ký tự phải xóa khỏi tên. Theo mặc định, các ký tự không hợp lệ là `\$%^&*()=+-\|[{';: /?.>,<`.

`label` Cung cấp danh sách các tên cần loại trừ, chẳng hạn như trả về, tệp, in. , tên đầu vào là `Nếu một trong những` một phần của danh sách này, ký tự gạch dưới (`_`) sẽ được thêm vào danh sách này.

`case_sensitive` Tên nên phân biệt chữ hoa chữ thường (`case_sensitive=True`), chuyển đổi thành chữ hoa (`case_sensitive=False` hoặc `case_sensitive='upper'`) hay thành chữ thường (`case_sensitive='low'`).

#### Tinh chỉnh chuyển đổi

##### Đổi số của bộ chuyển đổi

Thông thường, việc xác định một dtype là đủ để xác định cách chuyển đổi chuỗi chuỗi. Tuy nhiên, đôi khi có thể cần phải có một số biện pháp kiểm soát bổ sung. Ví dụ: chúng tôi có thể muốn đảm bảo rằng một ngày ở định dạng YYYY/MM/DD được chuyển đổi thành đối tượng `datetime` hoặc một chuỗi như `xx%` được chuyển đổi chính xác thành số float trong khoảng từ 0 đến 1.

Trong những trường hợp như vậy, chúng ta nên xác định các hàm chuyển đổi bằng các đổi số của bộ chuyển đổi.

Giá trị của đổi số này thường là một từ điển có chỉ mục cột hoặc tên cột làm khóa và hàm chuyển đổi làm giá trị. Các hàm chuyển đổi này có thể là hàm thực tế hoặc hàm lambda. Trong mọi trường hợp, họ chỉ nên chấp nhận một chuỗi làm đầu vào và đầu ra chỉ là một phần tử duy nhất của loại mong muốn.

Trong ví dụ sau, cột thứ hai được chuyển đổi từ dạng chuỗi biểu thị phần trăm thành số float trong khoảng từ 0 đến 1:

```
>>> Convertfunc = lambda x: float(x.strip(b"%"))/100. >>> dữ liệu = u"1,
2,3%, 45.\n6, 78,9%, 0" >>> tên = ("i", "p", "n")

>>> #Trường hợp chung.....
>>> np.genfromtxt(StringIO(data), delimiter="", name=names)
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
array([(1., nan, 45.), (6., nan, 0.)],
      dtype=[('i', '<f8'), ('p', '<f8'), ('n', '<f8')])
```

Chúng ta cần lưu ý rằng theo mặc định, `dtype=float`. Do đó, một float được mong đợi cho cột thứ hai.

Tuy nhiên, các chuỗi '2,3%' và '78,9%' không thể được chuyển đổi thành float và thay vào đó chúng ta có `np.nan`.

Bây giờ chúng ta hãy sử dụng một công cụ chuyển đổi:

```
>>> # Trường hợp được chuyển đổi ...
>>> np.genfromtxt(StringIO(data), delimiter=",", names=names, Converters={1: Convertfunc})
,
mảng([(1.0, 0.023, 45.0), (6.0, 0.7890000000000003, 0.0)], dtype=[('i',
      '<f8'), ('p', '<f8'), ('n', '<f8')])
```

Có thể thu được kết quả tương tự bằng cách sử dụng tên của cột thứ hai ("p") làm khóa thay vì chỉ mục của nó (1):

```
>>> # Sử dụng tên cho trình chuyển đổi ... >>>
np.genfromtxt(StringIO(data), delimiter=",", name=names,
      , bộ chuyển đổi={"p": Convertfunc})
mảng([(1.0, 0.023, 45.0), (6.0, 0.7890000000000003, 0.0)], dtype=[('i',
      '<f8'), ('p', '<f8'), ('n', '<f8')])
```

Bộ chuyển đổi cũng có thể được sử dụng để cung cấp giá trị mặc định cho các mục bị thiếu. Trong ví dụ sau, trình chuyển đổi biến đổi một chuỗi bị tước thành float tương ứng hoặc thành -999 nếu chuỗi trống. Chúng ta cần loại bỏ rõ ràng chuỗi khỏi khoảng trắng vì nó không được thực hiện theo mặc định:

```
data = u"1, , >>> chuyển 3\n 4, 5, 6" >>>
đổi = lambda x: float(x.strip() hoặc -999) >>> np.genfromtxt(StringIO(data),
dấu phân cách="", bộ chuyển đổi={1: chuyển đổi})
,
mảng([[ 1., -999., 5.,
      , 4., 6.]])
```

### Sử dụng các giá trị còn thiếu và điền

Một số mục có thể bị thiếu trong tập dữ liệu mà chúng tôi đang cố gắng nhập. Trong ví dụ trước, chúng ta đã sử dụng một trình chuyển đổi để chuyển đổi một chuỗi rỗng thành một chuỗi float. Tuy nhiên, các bộ chuyển đổi do người dùng xác định có thể nhanh chóng trở nên công kenne để quản lý.

Hàm `genfromtxt` cung cấp hai cơ chế bổ sung khác: đối số `loss_values` được sử dụng để nhận ra dữ liệu bị thiếu và đối số thứ hai, `fill_values`, được sử dụng để xử lý những dữ liệu bị thiếu này.

#### giá trị bị mất

Theo mặc định, bất kỳ chuỗi trống nào đều được đánh dấu là thiếu. Chúng ta cũng có thể xem xét các chuỗi phức tạp hơn, chẳng hạn như "Không có" hoặc "???" để thể hiện dữ liệu bị thiếu hoặc không hợp lệ. Đối số `lost_values` chấp nhận ba loại giá trị:

một chuỗi hoặc một chuỗi được phân tách bằng dấu phẩy Chuỗi này sẽ được sử dụng làm điểm đánh dấu cho dữ liệu bị thiếu cho tất cả các chuỗi cột

một chuỗi các chuỗi Trong trường hợp đó, mỗi mục được liên kết với một cột theo thứ tự.

a từ điển Giá trị của từ điển là các chuỗi hoặc chuỗi các chuỗi. Các khóa tương ứng có thể là chỉ mục cột (số nguyên) hoặc tên cột (chuỗi). Ngoài ra, phím đặc biệt Không có thể được sử dụng để xác định giá trị mặc định áp dụng cho tất cả các cột.

### diền giá trị

Chúng tôi biết cách nhận biết dữ liệu bị thiếu nhưng chúng tôi vẫn cần cung cấp giá trị cho các mục bị thiếu này. Theo mặc định, điều này giá trị được xác định từ dtype dự kiến theo bảng này:

Loại dự kiến	Mặc định
bool	SAI
int	-1
trôi nổi	np.nan
chuỗi phức	np.nan+0j
tập	,

Chúng ta có thể kiểm soát tốt hơn việc chuyển đổi các giá trị bị thiếu bằng đối số tùy chọn `fill_values`. Giống `thiếu_values`, đối số này chấp nhận các loại giá trị khác nhau:

một giá trị duy nhất Đây sẽ là giá trị mặc định cho tất cả các cột

một chuỗi giá trị Mỗi mục sẽ là mặc định cho cột tương ứng

một từ điển Mỗi khóa có thể là chỉ mục cột hoặc tên cột và giá trị tương ứng phải là

một đối tượng duy nhất. Chúng ta có thể sử dụng phím đặc biệt Không có để xác định giá trị mặc định cho tất cả các cột.

Trong ví dụ sau, chúng tôi giả sử rằng các giá trị bị thiếu được gắn cờ bằng "N/A" ở cột đầu tiên và bởi "?" trong cột thứ ba. Chúng tôi muốn chuyển đổi các giá trị còn thiếu này thành 0 nếu chúng xuất hiện ở cột thứ nhất và thứ hai, và tới -999 nếu chúng xuất hiện ở cột cuối cùng:

```
>>> data = u"N/A, 2, 3\n4, ,???"  
>>> kwargs = dict(delimiter=",",  
'              dtype=int,  
'              tên="a,b,c",  
'              thiếu_values={0:"N/A", 'b': "", 2:"???"},  
'              điền_values={0:0, 'b':0, 2:-999}  
>>> np.genfromtxt(StringIO(data), **kwargs)  
mảng([(0, 2, 3), (4, 0, -999)],  
      dtype=[('a', '<i8'), ('b', '<i8'), ('c', '<i8')])
```

### sử dụng mặt nạ

Chúng tôi cũng có thể muốn theo dõi sự xuất hiện của dữ liệu bị thiếu bằng cách xây dựng mặt nạ boolean với các mục nhập True, nơi dữ liệu bị thiếu và sai nếu không. Để làm điều đó, chúng ta chỉ cần đặt đối số tùy chọn `usemask` thành Đúng (mặc định là Sai). Mảng đầu ra sau đó sẽ là MaskedArray.

### chức năng phím tắt

Ngoài `genfromtxt`, mô-đun `numpy.lib.io` còn cung cấp một số chức năng tiện lợi bắt nguồn từ `genfromtxt`. Các hàm này hoạt động giống như hàm gốc nhưng chúng có các giá trị mặc định khác nhau.

`recfromtxt` Trả về một mảng `numpy.recarray` tiêu chuẩn (nếu `usemask=False`) hoặc một mảng `MaskedRecords` (nếu `usemask=True`). Dtype mặc định là `dtype=None`, nghĩa là loại của mỗi cột sẽ được xác định tự động.

`recfromcsv` Giống như `recfromtxt`, nhưng có dấu phân cách mặc định=",".

## 3.4 Lập chỉ mục

Xem thêm:

[lập chỉ mục](#)

[thủ tục lập chỉ mục](#)

Lập chỉ mục mảng để cập đến bất kỳ việc sử dụng dấu ngoặc vuông ([]) nào để lập chỉ mục các giá trị mảng. Có nhiều lựa chọn để lập chỉ mục, điều này mang lại sức mạnh to lớn cho việc lập chỉ mục gọn gàng, nhưng đi kèm với sức mạnh là một số sự phức tạp và khả năng gây nhầm lẫn. Phần này chỉ là tổng quan về các tùy chọn và vấn đề khác nhau liên quan đến việc lập chỉ mục. Ngoài việc lập chỉ mục phần tử đơn lẻ, bạn có thể tìm thấy chi tiết về hầu hết các tùy chọn này trong các phần liên quan.

### 3.4.1 Chuyển nhượng và tham chiếu

Hầu hết các ví dụ sau đây cho thấy việc sử dụng lập chỉ mục khi tham chiếu dữ liệu trong một mảng. Các ví dụ này cũng hoạt động tốt khi gán cho một mảng. Xem phần cuối để biết các ví dụ và giải thích cụ thể về cách thực hiện các bài tập.

### 3.4.2 Lập chỉ mục phần tử đơn

Lập chỉ mục phần tử đơn cho mảng 1-D là điều người ta mong đợi. Nó hoạt động chính xác như vậy đối với các chuỗi Python tiêu chuẩn khác. Nó dựa trên 0 và chấp nhận các chỉ số âm để lập chỉ mục từ cuối mảng.

```
>>> x = np.arange(10) >>> x[2]
2
>>> x[-2] 8
```

Không giống như danh sách và bộ dữ liệu, mảng có nhiều mảng hỗ trợ lập chỉ mục đa chiều cho mảng nhiều chiều. Điều đó có nghĩa là không cần thiết phải tách chỉ mục của từng thứ nguyên thành tập hợp dấu ngoặc vuông riêng.

```
>>> x.shape = (2,5) # bây giờ x là 2 chiều >>> x[1,3] 8

>>> x[1,-1] 9
```

Lưu ý rằng nếu một người lập chỉ mục một mảng nhiều chiều có ít chỉ số hơn thứ nguyên thì người ta sẽ nhận được một mảng nhiều chiều. Ví dụ:

```
>>> x[0]
mảng([0, 1, 2, 3, 4])
```

Nghĩa là, mỗi chỉ mục được chỉ định sẽ chọn mảng tương ứng với phần còn lại của kích thước đã chọn. Trong ví dụ trên, việc chọn 0 có nghĩa là chiều còn lại của độ dài 5 không được chỉ định và thứ được trả về là một mảng có chiều và kích thước đó. Cần lưu ý rằng mảng trả về không phải là bản sao của mảng gốc mà trả đến cùng các giá trị trong bộ nhớ giống như mảng ban đầu. Trong trường hợp này, mảng 1-D ở vị trí đầu tiên (0) được trả về. Vì vậy, việc sử dụng một chỉ mục duy nhất trên mảng được trả về sẽ dẫn đến một phần tử được trả về. Đó là:

```
>>> x[0][2] 2
```

Vì vậy, hãy lưu ý rằng  $x[0,2] = x[0][2]$  mặc dù trường hợp thứ hai kém hiệu quả hơn vì một mảng tạm thời mới được tạo sau chỉ mục đầu tiên được lập chỉ mục sau đó là 2.

Lưu ý những thứ được sử dụng theo thứ tự bộ nhớ IDL hoặc Fortran vì nó liên quan đến việc lập chỉ mục. NumPy sử dụng lập chỉ mục thứ tự C. Điều đó có nghĩa là chỉ mục cuối cùng thường biểu thị vị trí bộ nhớ thay đổi nhanh nhất, không giống như Fortran hoặc IDL, trong đó chỉ mục đầu tiên biểu thị vị trí thay đổi nhanh nhất trong bộ nhớ. Sự khác biệt này thể hiện khả năng gây nhầm lẫn lớn.

### 3.4.3 Các tùy chọn lập chỉ mục khác

Có thể cắt và sắp xếp các mảng để trích xuất các mảng có cùng số chiều nhưng có kích thước khác với mảng ban đầu. Việc cắt và kéo dài hoạt động giống hệt như cách thực hiện đổi với danh sách và bộ dữ liệu ngoại trừ việc chúng cũng có thể được áp dụng cho nhiều thứ nguyên. Một vài ví dụ minh họa rõ nhất:

```
>>> x = np.arange(10) >>>
x[2:5]
mảng([2, 3, 4]) >>>
x[:-7]
mảng([0, 1, 2]) >>>
x[1:7:2]
array([1, 3, 5]) >>>
y = np.arange(35).reshape(5,7) >>>
y[1:5:2,:,:3]
array([[ 7, 10 , 13], [21,
24, 27]])
```

Lưu ý rằng các lát mảng không sao chép dữ liệu mảng bên trong mà chỉ tạo ra các chế độ xem mới của dữ liệu gốc. Điều này khác với việc cắt danh sách hoặc bộ dữ liệu và nên sử dụng bản sao () rõ ràng nếu dữ liệu gốc không cần thiết nữa.

Có thể lập chỉ mục mảng với các mảng khác nhằm mục đích chọn danh sách các giá trị ngoài mảng vào mảng mới.

Có hai cách khác nhau để thực hiện điều này. Người ta sử dụng một hoặc nhiều mảng giá trị chỉ mục. Cách khác liên quan đến việc đưa ra một mảng boolean có hình dạng phù hợp để chỉ ra các giá trị được chọn. Mảng chỉ mục là một công cụ rất mạnh cho phép tránh lặp qua các phần tử riêng lẻ trong mảng và do đó cải thiện đáng kể hiệu suất.

Có thể sử dụng các tính năng đặc biệt để tăng số lượng kích thước trong một mảng một cách hiệu quả thông qua việc lập chỉ mục để mảng kết quả có được hình dạng cần thiết để sử dụng trong một biểu thức hoặc với một hàm cụ thể.

### 3.4.4 Mảng chỉ mục

Mảng NumPy có thể được lập chỉ mục với các mảng khác (hoặc bất kỳ đối tượng giống chuỗi nào khác có thể được chuyển đổi thành mảng, chẳng hạn như danh sách, ngoại trừ các bộ dữ liệu; hãy xem phần cuối của tài liệu này để biết lý do tại sao lại như vậy). Việc sử dụng mảng chỉ mục trái dài từ những trường hợp đơn giản, dễ hiểu đến những trường hợp phức tạp, khó hiểu. Đối với tất cả các trường hợp mảng chỉ mục, nội dung được trả về là bản sao của dữ liệu gốc, không phải dạng xem như dữ liệu có được đổi với các lát cắt.

Mảng chỉ mục phải có kiểu số nguyên. Mỗi giá trị trong mảng cho biết giá trị nào trong mảng sẽ được sử dụng thay cho chỉ mục. Để minh họa:

```
>>> x = np.arange(10,1,-1)
>>> x
array([10, 9, >>> 8, 7, 6, 5, 4, 3, 2])
x[np.array([3, 3, 1, 8])] array([7,
7, 9, 2])
```

Mảng chỉ mục gồm các giá trị 3, 3, 1 và 8 tương ứng tạo ra một mảng có độ dài 4 (giống như mảng chỉ mục) trong đó mỗi chỉ mục được thay thế bằng giá trị mà mảng chỉ mục có trong mảng được lập chỉ mục.

Các giá trị âm được cho phép và hoạt động giống như với các chỉ mục hoặc lát cắt đơn lẻ:

```
>>> x[np.array([3,3,-3,8])] array([7, 7,
4, 2])
```

Đó là một lỗi khi có giá trị chỉ mục nằm ngoài giới hạn:

```
>>> x[np.array([3, 3, 20, 8])] <type
'Exceptions.IndexError': chỉ số 20 ngoài giới hạn 0<=index<9
```

Nói chung, những gì được trả về khi sử dụng mảng chỉ mục là một mảng có hình dạng giống như mảng chỉ mục, nhưng có kiểu và giá trị của mảng được lập chỉ mục. Ví dụ: thay vào đó chúng ta có thể sử dụng mảng chỉ mục đa chiều:

```
>>> x[np.array([[1,1],[2,3]])] array([[9, 9],
[8, 7]])
```

### 3.4.5 Lập chỉ mục mảng đa chiều

Mọi thứ trở nên phức tạp hơn khi mảng đa chiều được lập chỉ mục, đặc biệt với mảng chỉ mục đa chiều. Những thứ này có xu hướng được sử dụng nhiều hơn nhưng chúng được cho phép và hữu ích trong một số vấn đề. Chúng ta sẽ bắt đầu với trường hợp đa chiều đơn giản nhất (sử dụng mảng y từ các ví dụ trước):

```
>>> y[np.array([0,2,4]), np.array([0,1,2])] array([ 0, 15, 30])
```

Trong trường hợp này, nếu các mảng chỉ mục có hình dạng trùng khớp và có một mảng chỉ mục cho từng chiều của mảng được lập chỉ mục thì mảng kết quả có hình dạng giống như mảng chỉ mục và các giá trị tương ứng với tập chỉ mục cho mỗi vị trí trong mảng chỉ mục. Trong ví dụ này, giá trị chỉ mục đầu tiên là 0 cho cả hai mảng chỉ mục và do đó giá trị đầu tiên của mảng kết quả là y[0,0]. Giá trị tiếp theo là y[2,1] và giá trị cuối cùng là y[4,2].

Nếu các mảng chỉ mục không có hình dạng giống nhau thì sẽ có nỗ lực phát chúng thành cùng một hình dạng. Nếu chúng không thể được phát theo cùng một hình dạng, một ngoại lệ sẽ được đưa ra:

```
>>> y[np.array([0,2,4]), np.array([0,1])] <type
'Exceptions.ValueError': hình dạng không khớp: các đối tượng không thể được phát sóng thành một hình
dạng duy nhất
```

Cơ chế phát sóng cho phép các mảng chỉ mục được kết hợp với các đại lượng vô hướng cho các chỉ mục khác. Hiệu quả là giá trị vô hướng được sử dụng cho tất cả các giá trị tương ứng của mảng chỉ số:

```
>>> y[np.array([0,2,4]), 1] array([ 1,
15, 29])
```

Chuyển sang mức độ phức tạp tiếp theo, chỉ có thể lập chỉ mục một phần một mảng với các mảng chỉ mục. Phải mất một chút suy nghĩ để hiểu điều gì xảy ra trong những trường hợp như vậy. Ví dụ: nếu chúng ta chỉ sử dụng một mảng chỉ mục với y:

```
>>> y[np.array([0,2,4])] array([[ 0,
1, 2, 3, 4, 5, 6], [14, 15, 16, 17, 18, 19, 20], [28,
29, 30, 31, 32, 33, 34]])
```

Kết quả là việc xây dựng một mảng mới trong đó mỗi giá trị của mảng chỉ mục chọn một hàng từ mảng được lập chỉ mục và mảng kết quả có hình dạng kết quả (số phần tử chỉ mục, kích thước của hàng).

Một ví dụ về nơi điều này có thể hữu ích là đối với bảng tra cứu màu nơi chúng tôi muốn ánh xạ các giá trị của hình ảnh thành bộ ba RGB để hiển thị. Bảng tra cứu có thể có hình dạng (nlookup, 3). Lập chỉ mục một mảng như vậy bằng một hình ảnh với

hình dạng (ny, nx) với dtype=np.uint8 (hoặc bất kỳ loại số nguyên nào miễn là các giá trị nằm trong giới hạn của bảng tra cứu) sẽ dẫn đến một mảng có hình dạng (ny, nx, 3) trong đó bộ ba của Giá trị RGB được liên kết với từng vị trí pixel.

Nói chung, hình dạng của mảng kết quả sẽ là sự kết hợp của hình dạng của mảng chỉ mục (hoặc hình dạng mà tất cả các mảng chỉ mục đã được phát triển) với hình dạng của bất kỳ kích thước nào không được sử dụng (những kích thước không được lập chỉ mục) trong mảng được lập chỉ mục . .

### 3.4.6 Mảng chỉ mục Boolean hoặc “mặt nạ”

Mảng Boolean được sử dụng làm chỉ mục được xử lý theo cách hoàn toàn khác với mảng chỉ mục. Mảng Boolean phải có hình dạng giống với kích thước ban đầu của mảng được lập chỉ mục. Trong trường hợp đơn giản nhất, mảng boolean có hình dạng giống nhau:

```
>>> b = y>20 >>>
y[b]
array([21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34])
```

Không giống như trường hợp mảng chỉ số nguyên, trong trường hợp boolean, kết quả là mảng 1-D chứa tất cả các phần tử trong mảng được lập chỉ mục tương ứng với tất cả các phần tử thực trong mảng boolean. Các phần tử trong mảng được lập chỉ mục luôn được lặp lại và trả về theo thứ tự hàng chính (kiểu C). Kết quả cũng giống với y[np.nonzero(b)]. Giống như mảng chỉ mục, những gì được trả về là một bản sao của dữ liệu, không phải là một dạng xem như với các lát cắt.

Kết quả sẽ là đa chiều nếu y có nhiều chiều hơn b. Ví dụ:

```
>>> b[:,5] # sử dụng boolean 1-D có độ mờ đầu tiên phù hợp với độ mờ đầu tiên của mảng y([False, False, False, True,
True]) >>> y[b[:,5]] mảng([[21, 22, 23, 24, 25, 26, 27],
[28, 29, 30, 31, 32, 33, 34]])
```

Ở đây, hàng thứ 4 và thứ 5 được chọn từ mảng được lập chỉ mục và kết hợp để tạo thành mảng 2-D.

Nói chung, khi mảng boolean có ít thứ nguyên hơn mảng được lập chỉ mục, điều này tương đương với y[b, . , , ], có nghĩa là y được lập chỉ mục bởi b, theo sau là số : cần thiết để điền vào thứ hạng của y. Do đó, hình dạng của kết quả là một chiều chứa số phần tử True của mảng boolean, theo sau là các chiều còn lại của mảng được lập chỉ mục.

Ví dụ: sử dụng mảng boolean 2-D có hình dạng (2,3) với bốn phần tử True để chọn các hàng từ mảng hình dạng 3-D (2,3,5) sẽ dẫn đến kết quả 2-D có hình dạng (4 ,5):

```
>>> x = np.arange(30).reshape(2,3,5)
>>> x
mảng([[[ 0, 1, 2, 3, 4], [ 5, 6, 7, 8, 9],
[10, 11, 12, 13, 14]],
[[15, 16, 17, 18, 19], [20, 21,
22, 23, 24], [25, 26, 27, 28,
29]]])
>>> b = np.array([[Đúng, Đúng, Sai], [Sai, Đúng, Đúng]]) >>> x[b] array([[ 0, 1, 2, 3,
4], [ 5,
6, 7, 8, 9], [20, 21, 22, 23, 24], [25,
26, 27, 28, 29]])
```

Để biết thêm chi tiết, hãy tham khảo tài liệu tham khảo gọn gàng về lập chỉ mục mảng.

### 3.4.7 Kết hợp mảng chỉ mục với các lát cắt

Mảng chỉ mục có thể được kết hợp với các lát cắt. Ví dụ:

```
>>> y[np.array([0,2,4]),1:3]
array([[ 1,  2],
       [15, 16],
       [29, 30]])
```

Trong thực tế, lát cắt được chuyển đổi thành một mảng chỉ mục np.array([[1,2]]) (hình dạng (1,2)) được phát cùng với mảng chỉ mục để tạo ra một mảng kết quả có hình dạng (3,2) .

Tương tự như vậy, việc cắt lát có thể được kết hợp với các chỉ số boolean được phát sóng:

```
>>> b = y > 20 >>
b
array([[Sai, Sai, Sai, Sai, Sai, Sai, Sai],
       [Sai, Sai, Sai, Sai, Sai, Sai, Sai],
       [Sai, Sai, Sai, Sai, Sai, Sai, Sai],
       [Đúng, Đúng, Đúng, Đúng, Đúng, Đúng, Đúng],
       [ Đúng, Đúng, Đúng, Đúng, Đúng, Đúng, Đúng]])
>>> y[b[:,5],1:3]
array([[22, 23],
       [29, 30]])
```

### 3.4.8 Công cụ lập chỉ mục cấu trúc

Để tạo điều kiện dễ dàng khớp các hình dạng mảng với các biểu thức và trong các phép gán, đối tượng np.newaxis có thể được sử dụng trong các chỉ mục mảng để thêm các thứ nguyên mới với kích thước là 1. Ví dụ:

```
>>> y.shape
(5,
7) >>> y[:,np.newaxis,:].shape
(5, 1, 7)
```

Lưu ý rằng không có phần tử mới nào trong mảng, chỉ có chiều được tăng lên. Điều này có thể hữu ích khi kết hợp hai mảng theo cách mà nếu không sẽ yêu cầu các hoạt động định hình lại một cách rõ ràng. Ví dụ:

```
>>> x = np.arange ( 5 )
>>> _, 3, 4, 5],
[2, 3, 4, 5, 6],
[3, 4, 5, 6, 7],
[4, 5, 6, 7, 8])
```

Cú pháp dấu chấm lửng có thể được sử dụng để biểu thị việc chọn đầy đủ mọi thứ nguyên chưa được chỉ định còn lại. Ví dụ:

```
>>> z = np.arange(81).reshape(3,3,3) >>
z[1,...,2]
array([[29, 32, 35],
       [38, 41, 44],
       [47, 50, 53]])
```

Điều này tương đương với:

```
>>> z[1,:,:2]
mảng([[29, 32, 35], [38, 41,
44], [47, 50, 53]])
```

### 3.4.9 Gán giá trị cho mảng được lập chỉ mục

Như đã đề cập, người ta có thể chọn một tập hợp con của một mảng để gán cho việc sử dụng một chỉ mục, các lát cắt, và các mảng chỉ mục và mặt nạ. Giá trị được gán cho mảng được lập chỉ mục phải có hình dạng nhất quán (cùng hình dạng hoặc có thể phát sóng theo hình dạng mà chỉ mục tạo ra). Ví dụ: được phép gán một hằng số cho một lát cắt:

```
>>> x = np.arange(10) >>>
x[2:7] = 1
```

hoặc một mảng có kích thước phù hợp:

```
>>> x[2:7] = np.arange(5)
```

Lưu ý rằng các phép gán có thể dẫn đến thay đổi nếu gán loại cao hơn cho loại thấp hơn (như float cho int) hoặc thậm chí ngoại lệ (gán phức tạp cho float hoặc int):

```
>>> x[1] = 1,2
>>> x[1]
1
>>> x[1] = 1.2j <type
'Exceptions.TypeError': không thể chuyển đổi phức tạp thành dài; sử dụng dài (abs(z))
```

Không giống như một số phép gán tham chiếu (chẳng hạn như chỉ mục mảng và mặt nạ) luôn được thực hiện đổi với dữ liệu gốc trong mảng (thực sự, không có gì khác có ý nghĩa!). Tuy nhiên, hãy lưu ý rằng một số hành động có thể không hoạt động như người ta mong đợi một cách ngây thơ. Ví dụ cụ thể này thường gây ngạc nhiên cho mọi người:

```
>>> x = np.arange(0, 50, 10)
>>> x
mảng([ 0, 10, 20, 30, 40]) >>>
x[np.array([1, 1, 3, 1])] += 1
>>> x
mảng([ 0, 11, 20, 31, 40])
```

Nơi mọi người kỳ vọng rằng vị trí đầu tiên sẽ được tăng thêm 3. Thực tế, nó sẽ chỉ được tăng thêm 1. Lý do là vì một mảng mới được trích xuất từ mảng gốc (dưới dạng tạm thời) chứa các giá trị tại 1, 1, 3, 1, thì giá trị 1 được thêm vào mảng tạm thời và sau đó giá trị tạm thời được gán trở lại mảng ban đầu. Do đó, giá trị của mảng tại x[1]+1 được gán cho x[1] ba lần, thay vì tăng lên 3 lần.

### 3.4.10 Xử lý số lượng chỉ số thay đổi trong chương trình

Cú pháp chỉ mục rất mạnh nhưng lại hạn chế khi xử lý số lượng chỉ mục thay đổi. Ví dụ: nếu bạn muốn viết một hàm có thể xử lý các đối số có số chiều khác nhau mà không cần phải viết mã trường hợp đặc biệt cho từng số chiều có thể, thì làm thế nào? Nếu người ta cung cấp cho chỉ mục một bộ, thì bộ đó sẽ được hiểu là một danh sách các chỉ mục. Ví dụ (sử dụng định nghĩa trước đó cho mảng z):

```
>>> chỉ số = (1,1,1,1) >>> z[chỉ
số] 40
```

Vì vậy, người ta có thể sử dụng mã để xây dựng các bộ dữ liệu có số lượng chỉ mục bất kỳ và sau đó sử dụng chúng trong một chỉ mục.

Các lát cắt có thể được chỉ định trong các chương trình bằng cách sử dụng hàm `slice()` trong Python. Ví dụ:

```
>>> chỉ số = (1,1,1,slice(0,2)) # giống như [1,1,1,0:2] >>> z[ chỉ số]
mảng([39, 40])
```

Tương tự như vậy, dấu ba chấm có thể được chỉ định bằng mã bằng cách sử dụng đối tượng Ellipsis:

```
>>> chỉ số = (1, Ellipsis, 1) # giống như [1,...,1] >>> z[ chỉ số] array([[28,
31, 34], [37, 40,
43], [46, 49, 52]])
```

Vì lý do này, bạn có thể sử dụng trực tiếp kết quả đầu ra từ hàm `np.nonzero()` làm chỉ mục vì nó luôn trả về một bộ mảng chỉ mục.

Bởi vì cách xử lý đặc biệt của các bộ dữ liệu, chúng không được tự động chuyển đổi thành một mảng như một danh sách. Như một ví dụ:

```
>>> z[[1,1,1,1]] # tạo ra một mảng lớn [[[27, 28, 29], [30, 31,
32], ...
>>> z[(1,1,1,1)] # trả về một giá trị
40
```

## 3.5 Phát sóng

Xem thêm:

`numpy.broadcast`

`array-broadcasting-in-numpy` Giới thiệu về các khái niệm được thảo luận ở đây

---

Lưu ý: Xem bài viết này để minh họa các khái niệm phát thanh truyền hình.

---

Thuật ngữ phát sóng mô tả cách xử lý các mảng có hình dạng khác nhau trong các phép tính số học. Tùy thuộc vào những ràng buộc nhất định, mảng nhỏ hơn sẽ được “phát” trên mảng lớn hơn để chúng có hình dạng tương thích.

Broadcasting cung cấp một phương tiện vector hóa các phép toán mảng để vòng lặp xảy ra trong C thay vì Python. Nó thực hiện điều này mà không tạo ra các bản sao dữ liệu không cần thiết và thường dẫn đến việc triển khai thuật toán hiệu quả. Tuy nhiên, có những trường hợp việc phát sóng là một ý tưởng tồi vì nó dẫn đến việc sử dụng bộ nhớ không hiệu quả và làm chậm quá trình tính toán.

Các thao tác NumPy thường được thực hiện trên các cặp mảng trên cơ sở từng phần tử. Trong trường hợp đơn giản nhất, hai mảng phải có hình dạng giống hệt nhau, như trong ví dụ sau:

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

Quy tắc phát sóng của NumPy nới lỏng ràng buộc này khi hình dạng của mảng đáp ứng các ràng buộc nhất định. Ví dụ phát sóng đơn giản nhất xảy ra khi một mảng và một giá trị vô hướng được kết hợp trong một thao tác:

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

Kết quả tương đương với ví dụ trước trong đó b là một mảng. Chúng ta có thể nghĩ rằng vô hướng b bị kéo dài trong quá trình thực hiện phép tính số học thành một mảng có hình dạng giống như a. Các phần tử mới trong b chỉ đơn giản là các bản sao của vô hướng ban đầu. Sự tương tự kéo dài chỉ mang tính khái niệm. NumPy đủ thông minh để sử dụng vô hướng ban đầu giá trị mà không thực sự tạo bản sao để các hoạt động phát sóng có hiệu quả về mặt bộ nhớ và tính toán như khả thi.

Mã trong ví dụ thứ hai hiệu quả hơn mã trong ví dụ đầu tiên vì việc phát sóng di chuyển ít bộ nhớ hơn trong quá trình nhân (b là vô hướng chứ không phải là mảng).

### 3.5.1 Quy tắc phát sóng chung

Khi hoạt động trên hai mảng, NumPy so sánh hình dạng của chúng theo từng phần tử. Nó bắt đầu với kích thước cuối cùng và hoạt động theo cách của nó về phía trước. Hai chiều tương thích khi

- 1) chúng bằng nhau, hoặc
- 2) một trong số đó là 1

Nếu những điều kiện này không được đáp ứng, thì một lỗi `ValueError`: toán hạng không thể được phát cùng nhau sẽ được đưa ra ngoại lệ, cho biết rằng các mảng có hình dạng không tương thích. Kích thước của mảng kết quả là kích thước không phải 1 dọc theo mỗi trục của đầu vào.

Mảng không cần phải có cùng số chiều. Ví dụ: nếu bạn có mảng RGB 256x256x3 value và bạn muốn chia tỷ lệ từng màu trong ảnh theo một giá trị khác nhau, bạn có thể nhân hình ảnh với mảng một chiều có 3 giá trị. Sắp xếp kích thước các trục sau của các mảng này theo kích thước phát sóng quy tắc, cho thấy rằng chúng tương thích:

```
Hình ảnh ( mảng 3d): 256 x 256 x 3
Tỷ lệ ( mảng 1d): 3
Kết quả ( mảng 3d): 256 x 256 x 3
```

Khi một trong hai kích thước được so sánh là một thì kích thước kia sẽ được sử dụng. Nói cách khác, kích thước có kích thước 1 được kéo dài hoặc "sao chép" để khớp với cái khác.

Trong ví dụ sau, cả mảng A và B đều có trục có độ dài bằng 1 và được mở rộng đến kích thước lớn hơn trong quá trình hoạt động phát sóng:

```
MỘT      ( Mảng 4d): 8 x 1 x 6 x 1
B        ( mảng 3d): 7 x 1 x 5
Kết quả ( mảng 4d): 8 x 7 x 6 x 5
```

Dưới đây là một số ví dụ:

```
MỘT      ( Mảng 2d): 5 x 4
B        ( mảng 1d): 1
Kết quả ( mảng 2d): 5 x 4

MỘT      ( Mảng 2d): 5 x 4
B        ( mảng 1d): 4
Kết quả ( mảng 2d): 5 x 4

MỘT      ( mảng 3d): 15 x 3 x 5
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
B          ( mảng 3d): 15 x 1 x 5
Kết quả ( mảng 3d): 15 x 3 x 5

MỘT      ( mảng 3d): 15 x 3 x 5
B          ( Mảng 2d): 3 x 5
Kết quả ( mảng 3d): 15 x 3 x 5

MỘT      ( mảng 3d): 15 x 3 x 5
B          ( Mảng 2d): 3 x 1
Kết quả ( mảng 3d): 15 x 3 x 5
```

Dưới đây là ví dụ về các hình dạng không phát sóng:

```
MỘT      ( mảng 1d): 3
B          ( Mảng 1d): 4 # kích thước cuối không khớp

MỘT      ( Mảng 2d):           2x1 _ _
B          ( Mảng 3d ): 8 x 4 x 3 # giây tính từ kích thước cuối cùng không khớp
```

Một ví dụ về phát sóng trong thực tế:

```
>>> x = np.arange(4)
>>> xx = x.reshape(4,1)
>>> y = np.ones(5)
>>> z = np.ones((3,4))

>>> x.hình dạng
(4,)

>>> y.hình dạng
(5,)

>>> x + y
ValueError: toán hạng không thể được phát cùng với hình dạng (4,) (5,)

>>> xx.hình dạng
(4, 1)

>>> y.hình dạng
(5,)

>>> (xx + y).hình dạng
(4, 5)

>>> xx + y
mảng([[ 1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.]])
```

```
>>> x.hình dạng
(4,)

>>> z.hình dạng
(3, 4)

>>> (x + z).hình dạng
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
(3, 4)

>>> x + z
mảng([[ 1., 2., 3., 4.], [ 1., 2.,
       3., 4.], [ 1., 2., 3., 4.]])
```

Broadcasting cung cấp một cách thuận tiện để lấy sản phẩm bên ngoài (hoặc bất kỳ hoạt động bên ngoài nào khác) của hai mảng. Ví dụ sau đây cho thấy phép cộng bên ngoài của hai mảng 1-d:

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

Ở đây, toán tử chỉ mục trực mới sẽ chèn một trực mới vào a, biến nó thành một mảng 4x1 hai chiều. Kết hợp mảng 4x1 với b, có hình dạng (3,), tạo ra mảng 4x3.

## 3.6 Hoán đổi byte

### 3.6.1 Giới thiệu về thứ tự byte và ndarray

Ndarray là một đối tượng cung cấp giao diện mảng python cho dữ liệu trong bộ nhớ.

Điều thường xảy ra là bộ nhớ mà bạn muốn xem bằng một mảng không có cùng thứ tự byte với máy tính mà bạn đang chạy Python.

Ví dụ: tôi có thể đang làm việc trên một máy tính có CPU endian nhỏ - chẳng hạn như Intel Pentium, nhưng tôi đã tải một số dữ liệu từ một tệp được viết bởi một máy tính có endian lớn. Giả sử tôi đã tải 4 byte từ một tệp được viết bởi máy tính Sun (big endian). Tôi biết rằng 4 byte này đại diện cho hai số nguyên 16 bit. Trên máy tính lớn, một số nguyên hai byte được lưu trữ với Byte quan trọng nhất (MSB) trước tiên và sau đó là Byte quan trọng nhất (LSB). Do đó, các byte theo thứ tự bộ nhớ là:

1. Số nguyên MSB 1
2. LSB số nguyên 1
3. Số nguyên MSB 2
4. LSB số nguyên 2

Giả sử hai số nguyên thực tế là 1 và 770. Vì  $770 = 256 * 3 + 2$  nên 4 byte trong bộ nhớ sẽ lần lượt chứa: 0, 1, 3, 2. Các byte tôi đã tải từ tệp sẽ có những nội dung này . ::

```
>>> big_end_buffer = bytearray([0,1,3,2]) >>>
big_end_buffer
bytearray(b'\x00\x01\x03\x02')
```

Chúng ta có thể muốn sử dụng ndarray để truy cập các số nguyên này. Trong trường hợp đó, chúng ta có thể tạo một mảng xung quanh bộ nhớ này và nói với Numpy rằng có hai số nguyên và chúng là 16 bit và big-endian:

```
>>> nhập numpy dưới dạng np
>>> big_end_arr = np.ndarray(shape=(2,), dtype='>i2', buffer=big_end_buffer) >>> big_end_arr[0]
1
>>> big_end_arr[1] 770
```

Lưu ý mảng dtype ở trên >i2. > có nghĩa là 'big-endian' (< là little-endian) và i2 có nghĩa là 'số nguyên 2 byte có dấu'. Ví dụ: nếu dữ liệu của chúng tôi biểu thị một số nguyên cuối nhò 4 byte không dấu, thì chuỗi dtype sẽ là <u4.

Thực tế, tại sao chúng ta không thử điều đó?

```
>>> little_end_u4 = np.ndarray(shape=(1,), dtype='<u4', buffer=big_end_buffer) >>> little_end_u4[0]
== 1 * 256**1 + 3 * 256**2 + 2 * 256**3
Đúng vậy
```

Quay lại big\_end\_arr của chúng tôi - trong trường hợp này, dữ liệu cơ bản của chúng tôi là big-endian (độ cuối của dữ liệu) và chúng tôi đã đặt dtype khớp (dtype cũng là big-endian). Tuy nhiên, đôi khi bạn cần phải lật lại những thứ này.

Cảnh báo: Hiện tại, số vô hướng không bao gồm thông tin thứ tự byte, do đó việc trích xuất số vô hướng từ một mảng sẽ trả về một số nguyên theo thứ tự byte gốc. Kể từ đây:

```
>>> big_end_arr[0].dtype.byteorder == little_end_u4[0].dtype.byteorder Đúng
```

### 3.6.2 Thay đổi thứ tự byte

Như bạn có thể tưởng tượng từ phần giới thiệu, có hai cách bạn có thể tác động đến mối quan hệ giữa thứ tự byte của mảng và bộ nhớ cơ bản mà nó đang xem xét:

- Thay đổi thông tin thứ tự byte trong mảng dtype để nó diễn giải dữ liệu cơ bản như trong một thứ tự byte khác nhau. Đây là vai trò của arr.newbyteorder()
- Thay đổi thứ tự byte của dữ liệu cơ bản, giữ nguyên cách diễn giải dtype. Đây là những gì arr.byteswap() thì có.

Các tình huống phổ biến mà bạn cần thay đổi thứ tự byte là:

1. Dữ liệu và độ bền của dtype của bạn không khớp và bạn muốn thay đổi dtype sao cho khớp với dữ liệu.
2. Dữ liệu của bạn và độ bền dtype không nhau và bạn muốn hoán đổi dữ liệu sao cho chúng khớp với dtype.
3. Dữ liệu của bạn và độ bền của dtype khớp nhau, nhưng bạn muốn dữ liệu được hoán đổi và dtype phản ánh điều này.

Độ bền của dữ liệu và dtype không khớp, hãy thay đổi dtype để khớp với dữ liệu

Chúng tôi tạo ra thứ gì đó mà chúng không khớp:

```
>>> sai_end_dtype_arr = np.ndarray(shape=(2,), dtype='<i2', buffer=big_end_buffer) >>> sai_end_dtype_arr[0]
256
```

Cách khắc phục rõ ràng cho tình huống này là thay đổi dtype để nó mang lại độ bền chính xác:

```
>>> fix_end_dtype_arr = sai_end_dtype_arr.newbyteorder() >>>
fix_end_dtype_arr[0]
1
```

Lưu ý mảng không thay đổi trong bộ nhớ:

```
>>> fix_end_dtype_arr.tobytes() == big_end_buffer
ĐÚNG VẬY
```

Độ bền của dữ liệu và kiểu không khớp, thay đổi dữ liệu để khớp với dtype

Bạn có thể muốn thực hiện việc này nếu bạn cần dữ liệu trong bộ nhớ theo một thứ tự nhất định. Ví dụ: bạn có thể ghi bộ nhớ ra một tệp cần thứ tự byte nhất định.

```
>>> fix_end_mem_arr = sai_end_dtype_arr.byteswap() >>>
fix_end_mem_arr[0] 1
```

Bây giờ mảng đã thay đổi trong bộ nhớ:

```
>>> fix_end_mem_arr.tobytes() == big_end_buffer Sai
```

Độ bền cuối cùng của dữ liệu và dtype, trao đổi dữ liệu và dtype

Bạn có thể có một dtype mảng được chỉ định chính xác, nhưng bạn cần mảng có thứ tự byte ngược lại trong bộ nhớ và bạn muốn dtype khớp với nhau để các giá trị mảng có ý nghĩa. Trong trường hợp này, bạn chỉ cần thực hiện cả hai thao tác trước đó:

```
>>> swapped_end_arr = big_end_arr.byteswap().newbyteorder() >>>
swapped_end_arr[0]
1
>>> swapped_end_arr.tobytes() == big_end_buffer Sai
```

Có thể thực hiện được cách dễ dàng hơn để truyền dữ liệu tới một thứ tự dtype và byte cụ thể bằng phương thức ndarray astype:

```
>>> swapped_end_arr = big_end_arr.astype('<i2') >>>
swapped_end_arr[0] 1
>>> swapped_end_arr.tobytes() == big_end_buffer Sai
```

## 3.7 Mảng có cấu trúc

### 3.7.1 Giới thiệu

Mảng có cấu trúc là các mảng có kiểu dữ liệu là sự kết hợp của các kiểu dữ liệu đơn giản hơn được tổ chức dưới dạng một chuỗi các trường được đặt tên. Ví dụ,

```
>>> x = np.array([('Rex', 9, 81.0), ('Fido', 3, 27.0)],
   dtype=[('tên', 'U10'), ('tuổi', 'i4'), ('cân nặng', 'f4')])
>>> x
array([('Rex', 9, 81.), ('Fido', 3, 27.)],
      dtype=[('name', 'U10'), ('age', '<i4'), ('trọng lượng', '<f4')])
```

Ở đây x là mảng một chiều có độ dài hai có kiểu dữ liệu là cấu trúc có ba trường: 1. Một chuỗi có độ dài từ 10 trở xuống có tên là 'name', 2. một số nguyên 32 bit có tên là 'age' và 3. một số thực 32 bit có tên là 'trọng lượng'.

Nếu bạn lặp chỉ mục x ở vị trí 1, bạn sẽ nhận được cấu trúc:

```
>>> x[1]
("Fido", 3, 27.0)
```

Bạn có thể truy cập và sửa đổi các trường riêng lẻ của một mảng có cấu trúc bằng cách lập chỉ mục với tên trường:

```
>>> x['age']
mảng([9, 3], dtype=int32) >>>
x['age'] = 5
>>> x
array([('Rex', 5, 81.), ('Fido', 5, 27.)],
      dtype=[('name', 'U10'), ('age', '<i4'), ('trọng lượng', '<f4')])
```

Các kiểu dữ liệu có cấu trúc được thiết kế để có thể bắt chước 'cấu trúc' trong ngôn ngữ C và chia sẻ bộ nhớ tương tự.

Chúng nhằm mục đích giao tiếp với mã C và dễ dàng trao đổi với các bộ đệm có cấu trúc, ví dụ như để diễn giải các đốm màu nhị phân. Với những mục đích này, chúng hỗ trợ các tính năng chuyên dụng như mảng con, kiểu dữ liệu lồng nhau và liên kết, đồng thời cho phép kiểm soát bộ nhớ của cấu trúc.

Người dùng đang tìm cách thao tác dữ liệu dạng bảng, chẳng hạn như được lưu trữ trong tệp csv, có thể tìm thấy các dự án pydata khác phù hợp hơn, chẳng hạn như xarray, pandas hoặc DataArray. Chúng cung cấp giao diện cấp cao để phân tích dữ liệu dạng bảng và được tối ưu hóa tốt hơn cho mục đích sử dụng đó. Ví dụ: bộ đệm bộ nhớ giống cấu trúc C của các mảng có cấu trúc ở dạng gọn gàng có thể dẫn đến hoạt động của bộ đệm kém khi so sánh.

### 3.7.2 Kiểu dữ liệu có cấu trúc

Kiểu dữ liệu có cấu trúc có thể được coi là một chuỗi byte có độ dài nhất định (kích thước mục của cấu trúc) được hiểu là một tập hợp các trường. Mỗi trường có một tên, kiểu dữ liệu và độ lệch byte trong cấu trúc. Kiểu dữ liệu của một trường có thể là bất kỳ kiểu dữ liệu gọn gàng nào bao gồm các kiểu dữ liệu có cấu trúc khác và nó cũng có thể là kiểu dữ liệu mảng con hoạt động giống như một mảng ndarray có hình dạng được chỉ định. Độ lệch của các trường là tùy ý và các trường thậm chí có thể chồng lên nhau. Những độ lệch này thường được xác định tự động bằng numpy, nhưng cũng có thể được chỉ định.

#### Tạo kiểu dữ liệu có cấu trúc

Các kiểu dữ liệu có cấu trúc có thể được tạo bằng hàm `numpy.dtype`. Có 4 dạng đặc tả thay thế khác nhau về tính linh hoạt và ngắn gọn. Những điều này được ghi lại thêm trong trang tham chiếu Đối tượng Kiểu Dữ liệu và tóm tắt chúng là:

##### 1. Danh sách các bộ, mỗi bộ một trường

Mỗi bộ dữ liệu có dạng (tên trường, kiểu dữ liệu, hình dạng) trong đó hình dạng là tùy chọn. tên trường là một chuỗi (hoặc bộ dữ liệu nếu tiêu đề được sử dụng, xem Tiêu đề trường bên dưới), kiểu dữ liệu có thể là bất kỳ đối tượng nào có thể chuyển đổi thành kiểu dữ liệu và hình dạng là một bộ số nguyên chỉ định hình dạng mảng con.

```
>>> np.dtype([('x', 'f4'), ('y', np.float32), ('z', 'f4', (2, 2))]) dtype([('x',
   '<f4'), ('y', '<f4'), ('z', '<f4', (2, 2))])
```

Nếu tên trường là chuỗi rỗng '' thì trường sẽ có tên mặc định có dạng f#, trong đó # là chỉ số nguyên của trường, tính từ 0 từ bên trái:

```
>>> np.dtype([('x', 'f4'), ('', 'i4'), ('z', 'i8')]) dtype([('x', '<f4'), ('f1', '<i4'), ('z', '<i8')])
```

Độ lệch byte của các trường trong cấu trúc và tổng kích thước mục của cấu trúc được xác định tự động.

## 2. Một chuỗi thông số kỹ thuật dtype được phân tách bằng dấu phẩy

Trong ký hiệu viết tắt này, bất kỳ thông số kỹ thuật dtype nào của chuỗi đều có thể được sử dụng trong một chuỗi và được phân tách bằng dấu phẩy. Kích thước mục và độ lệch byte của các trường được xác định tự động và tên trường được đặt tên mặc định là f0, f1, v.v.

```
>>> np.dtype('i8, f4, S3')
dtype([('f0', '<i8'), ('f1', '<f4'), ('f2', 'S3')]) >>>
np.dtype('3int8, float32, (2, 3)float64') dtype([('f0', 'i1', (3,)), ('f1', '<f4'), ('f2', '<f8', (2, 3))])
```

## 3. Từ điển mảng tham số trường

Đây là dạng đặc tả linh hoạt nhất vì nó cho phép kiểm soát độ lệch byte của các trường và kích thước mục của cấu trúc.

Từ điển có hai khóa bắt buộc là 'tên' và 'định dạng' và bốn khóa tùy chọn là 'offsets', 'itemsize', 'aligned' và 'titles'.

Các giá trị cho 'tên' và 'định dạng' lần lượt phải là danh sách tên trường và danh sách thông số dtype, có cùng độ dài.

Giá trị 'độ lệch' tùy chọn phải là danh sách các độ lệch byte số nguyên, một giá trị cho mỗi trường trong cấu trúc. Nếu 'bù đắp' không được đưa ra thì các khoản bù trừ sẽ được xác định tự động.

Giá trị 'itemsize' tùy chọn phải là một số nguyên mô tả tổng kích thước tính bằng byte của dtype, phải đủ lớn để chứa tất cả các trường.

```
>>> np.dtype({'names': ['col1', 'col2'], 'formats': ['i4', 'f4']}) dtype([('col1', '<i4'), ('col2', '<f4')])
>>> np.dtype({'names': ['col1', 'col2'], 'formats': ['i4', 'f4'], 'offsets': [0, 4], 'itemsize': 12}) dtype({'names': ['col1', 'col2'], 'formats': ['<i4', '<f4'], 'offset': [0, 4], 'itemsize': 12})
```

Độ lệch có thể được chọn sao cho các trường trùng nhau, mặc dù điều này có nghĩa là việc gán cho một trường có thể ghi đè bất kỳ dữ liệu nào của trường chồng chéo. Là một ngoại lệ, các trường thuộc loại numpy.object không thể trùng lặp với các trường khác, do có nguy cơ ghi đè con trỏ đối tượng bên trong và sau đó hủy tham chiếu nó.

Giá trị 'căn chỉnh' tùy chọn có thể được đặt thành True để làm cho việc tính toán độ lệch tự động sử dụng độ lệch được căn chỉnh (xem Độ lệch và căn chỉnh byte tự động), như thể đối số từ khóa 'căn chỉnh' của numpy.dtype đã được đặt thành True.

Giá trị 'tiêu đề' tùy chọn phải là danh sách các tiêu đề có cùng độ dài với 'tên', xem Tiêu đề trường bên dưới.

## 4. Từ điển tên trường

Việc sử dụng dạng đặc tả này không được khuyến khích nhưng được ghi lại ở đây vì mã numpy cũ hơn có thể sử dụng nó. Các khóa của từ điển là tên trường và các giá trị là các bộ dữ liệu chỉ định loại và offset:

```
>>> np.dtype({'col1': ('i1', 0), 'col2': ('f4', 1)})
dtype([('col1', 'i1'), ('col2', '<f4')])
```

Biểu mẫu này không được khuyến khích vì từ điển Python không giữ nguyên thứ tự trong các phiên bản Python trước Python 3.6 và thứ tự của các trường trong một dtype có cấu trúc có ý nghĩa. Tiêu đề trường có thể được chỉ định bằng cách sử dụng một

3-tuple, xem bên dưới.

Thao tác và hiển thị các kiểu dữ liệu có cấu trúc

Danh sách tên trường của kiểu dữ liệu có cấu trúc có thể được tìm thấy trong thuộc tính tên của đối tượng dtype:

```
>>> d = np.dtype([('x', 'i8'), ('y', 'f4')]) >>> d.names
('x', 'y')
```

Tên trường có thể được sửa đổi bằng cách gán cho thuộc tính tên bằng cách sử dụng một chuỗi các chuỗi có cùng độ dài.

Đối tượng dtype cũng có một thuộc tính giống như từ điển, các trường, có khóa là tên trường (và [Tiêu đề trường](#), xem bên dưới) và có giá trị là các bộ chứa dtype và byte offset của mỗi trường.

```
>>> d.fields
mapsproxy({'x': (dtype('int64'), 0), 'y': (dtype('float32'), 8)})
```

Cả thuộc tính tên và trường sẽ bằng Không đổi với mảng không có cấu trúc. Cách được khuyến nghị để kiểm tra xem một dtype có được cấu trúc hay không là sử dụng if dt.names không phải là None thay vì if dt.names, để giải thích cho các dtype có 0 trường.

Biểu diễn chuỗi của kiểu dữ liệu có cấu trúc được hiển thị ở dạng “danh sách các bộ dữ liệu” nếu có thể, nếu không thì sẽ quay trở lại sử dụng biểu mẫu từ điển tổng quát hơn.

Tự động bù đắp và căn chỉnh byte

Numpy sử dụng một trong hai phương pháp để tự động xác định độ lệch byte của trường và kích thước tổng thể của một kiểu dữ liệu có cấu trúc, tùy thuộc vào việc căn chỉnh=True có được chỉ định làm đối số từ khóa cho numpy.dtype hay không.

Theo mặc định (align=False), numpy sẽ đóng gói các trường lại với nhau sao cho mỗi trường bắt đầu ở byte bù mà trường trước đó đã kết thúc và các trường liền kề nhau trong bộ nhớ.

```
>>> def print_offsets(d):
...     print("offsets:", [d.fields[name][1] cho tên trong d.names])
...     print("itemsize:", d.itemsize)
>>> print_offsets(np.dtype('u1, u1, i4, u1, i8, u2')) offset: [0,
1, 2, 6, 7, 15] kích thước mục: 17
```

Nếu căn chỉnh=True được đặt, numpy sẽ đệm cấu trúc giống như cách mà nhiều trình biên dịch C sẽ đệm cấu trúc C. Các cấu trúc được căn chỉnh có thể cải thiện hiệu suất trong một số trường hợp, nhưng phải trả giá bằng việc tăng kích thước kiểu dữ liệu. Các byte đệm được chèn vào giữa các trường sao cho độ lệch byte của mỗi trường sẽ là bội số của căn chỉnh của trường đó, thường bằng kích thước của trường tính bằng byte đối với các kiểu dữ liệu đơn giản, xem [PyArray\\_Descr.alignment](#). Cấu trúc cũng sẽ được thêm phần đệm cuối để kích thước các mục của nó là bội số của căn chỉnh của trường lớn nhất.

```
>>> print_offsets(np.dtype('u1, u1, i4, u1, i8, u2', align=True)) offset: [0, 1,
4, 8, 16, 24] kích thước mục: 32
```

Lưu ý rằng mặc dù hầu hết tất cả các trình biên dịch C hiện đại đều đệm theo cách này theo mặc định, nhưng phần đệm trong cấu trúc C phụ thuộc vào việc triển khai C nên bối cảnh này không được đảm bảo khớp chính xác với cấu trúc tương ứng trong chương trình C.

Có thể cần một số công việc, ở phía khó khăn hoặc phía C, để có được sự tương ứng chính xác.

Nếu độ lệch được chỉ định bằng cách sử dụng khóa offset tùy chọn trong đặc tả dtype dựa trên từ điển, cài đặt căn chỉnh=True sẽ kiểm tra xem độ lệch của mỗi trường có phải là bội số của kích thước của nó không và kích thước mục là bội số của kích thước trường lớn nhất và đưa ra một ngoại lệ . nếu không.

Nếu độ lệch của các trường và kích thước mục của một mảng có cấu trúc thỏa mãn các điều kiện căn chỉnh thì mảng đó sẽ có cờ CĂN HỘ được đặt.

Hàm tiện lợi `numpy.lib.recykling_fields` chuyển đổi một kiểu hoặc mảng được căn chỉnh thành một kiểu được đóng gói và ngược lại. Nó lấy một `dtype` hoặc `ndarray` có cấu trúc làm đối số và trả về một bản sao với các trường được đóng gói lại, có hoặc không có byte đệm.

#### Tiêu đề trường

Ngoài tên trường, các trường cũng có thể có tiêu đề liên quan, tên thay thế, đôi khi được sử dụng làm mô tả bổ sung hoặc bí danh cho trường. Tiêu đề có thể được sử dụng để lập chỉ mục cho một mảng, giống như tên trường.

Để thêm tiêu đề khi sử dụng dạng danh sách bộ dữ liệu của đặc tả `dtype`, tên trường có thể được chỉ định dưới dạng bộ gồm hai chuỗi thay vì một chuỗi, tương ứng sẽ là tiêu đề của trường và tên trường. Ví dụ:

```
>>> np.dtype([('tiêu đề của tôi', 'tên'), ('f4')]) dtype([('tiêu  
đề của tôi', 'tên'), ('<f4')])
```

Khi sử dụng dạng đặc tả dựa trên từ điển đầu tiên, tiêu đề có thể được cung cấp dưới dạng khóa 'tiêu đề' bổ sung như được mô tả ở trên. Khi sử dụng đặc tả dựa trên từ điển thứ hai (không được khuyến khích), tiêu đề có thể được cung cấp bằng cách cung cấp bộ dữ liệu 3 phần tử (kiểu dữ liệu, offset, tiêu đề) thay vì bộ dữ liệu 2 phần tử thông thường:

```
>>> np.dtype({'name': ('i4', 0, 'my title')})  
dtype([('my title', 'name'), ('<i4')])
```

Từ điển `dtype.fields` sẽ chứa các tiêu đề làm khóa, nếu có bất kỳ tiêu đề nào được sử dụng. Điều này có nghĩa là một trường có tiêu đề sẽ được hiển thị hai lần trong từ điển trường. Các giá trị tuple cho các trường này cũng sẽ có phần tử thứ ba là tiêu đề trường. Vì điều này và vì thuộc tính tên duy trì thứ tự trường trong khi thuộc tính trường có thể không, nên lặp qua các trường của `dtype` bằng cách sử dụng thuộc tính tên của `dtype`, thuộc tính này sẽ không liệt kê tiêu đề, như trong:

```
>>> cho tên trong d.names:  
,     print(d.fields[name][:2])  
(dtype('int64'), 0)  
(dtype('float32'), 8)
```

các loại công đoạn

Các kiểu dữ liệu có cấu trúc được triển khai theo kiểu numpy để có kiểu cơ sở numpy.void theo mặc định, nhưng có thể hiểu các kiểu numpy khác là kiểu có cấu trúc bằng cách sử dụng dạng (`base_dtype`, `dtype`) của đặc tả `dtype` được mô tả trong `Đổi tượng kiểu dữ liệu`. Ở đây, `base_dtype` là `dtype` cơ bản mong muốn, đồng thời các trường và cờ sẽ được sao chép từ `dtype`. `Dtype` này tương tự như 'union' trong C.

### 3.7.3 Lập chỉ mục và gán cho mảng có cấu trúc

#### Gán dữ liệu vào một mảng có cấu trúc

Có một số cách để gán giá trị cho một mảng có cấu trúc: Sử dụng bộ dữ liệu python, sử dụng giá trị vô hướng hoặc sử dụng các mảng có cấu trúc khác.

### Bài tập từ các kiểu gốc Python (Bộ dữ liệu)

Cách đơn giản nhất để gán giá trị cho một mảng có cấu trúc là sử dụng bộ dữ liệu python. Mỗi giá trị được gán phải là một bộ có độ dài bằng số trường trong mảng chứ không phải là danh sách hoặc mảng vì những giá trị này sẽ kích hoạt quy tắc phát sóng của numpy. Các phần tử của bộ dữ liệu được gán cho các trường liên tiếp của mảng, từ trái sang phải:

```
>>> x = np.array([(1, 2, 3), (4, 5, 6)], dtype='i8, f4, f8') >>> x[1] =
(7, 8, 9)
>>> x
mảng([(1, 2., 3.), (7, 8., 9.)],
      dtype=[('f0', '<i8'), ('f1', '<f4'), ('f2', '<f8')])
```

### Bài tập từ vô hướng

Một đại lượng vô hướng được gán cho một phần tử có cấu trúc sẽ được gán cho tất cả các trường. Điều này xảy ra khi một đại lượng vô hướng được gán cho một mảng có cấu trúc hoặc khi một mảng không có cấu trúc được gán cho một mảng có cấu trúc:

```
>>> x = np.zeros(2, dtype='i8, f4, ?, S1') >>> x[:, :
= 3
>>> x
mảng([(3, 3., Đúng, b'3'), (3, 3., Đúng, b'3')],
      dtype=[('f0', '<i8'), ('f1', '<f4'), ('f2', '?'), ('f3', 'S1')]) >>> x[:, :] =
np.arange(2)
>>> x
array([(0, 0., False, b'0'), (1, 1., True, b'1')], dtype=[('f0',
      '<i8'), ('f1', '<f4'), ('f2', '?'), ('f3', 'S1')))
```

Mảng có cấu trúc cũng có thể được gán cho mảng không có cấu trúc, nhưng chỉ khi kiểu dữ liệu có cấu trúc chỉ có một trường duy nhất:

```
>>> twofield = np.zeros(2, dtype=[('A', 'i4'), ('B', 'i4')]) >>> onefield = np.zeros(2,
dtype=[('A', 'i4')])
>>> nostruct = np.zeros(2, dtype='i4')
nostruct[:] = twofield
Traceback (cuộc gọi gần đây nhất
gần đây nhất):
'
TypeError: Không thể truyền vô hướng từ dtype([('A', '<i4'), ('B', '<i4')]) sang dtype('int32') theo quy tắc 'không an toàn'
```

### Chuyển nhượng từ các mảng có cấu trúc khác

Việc gán giữa hai mảng có cấu trúc xảy ra như thế các phần tử nguồn đã được chuyển đổi thành các bộ dữ liệu và sau đó được gán cho các phần tử đích. Nghĩa là, trường đầu tiên của mảng nguồn được gán cho trường đầu tiên của mảng đích và trường thứ hai cũng vậy, v.v., bắt kè tên trường. Các mảng có cấu trúc với số lượng trường khác nhau không thể được gán cho nhau. Các byte của cấu trúc đích không được bao gồm trong bất kỳ trường nào sẽ không bị ảnh hưởng.

```
>>> a = np.zeros(3, dtype=[('a', 'i8'), ('b', 'f4'), ('c', 'S3')]) >>> b =
np.ones(3, dtype=[('x', 'f4'), ('y', 'S3'), ('z', 'O')]) >>> b[:] = a >>> b
array([(0., b'0.0', b''), (0., b'0.0', b''), (0., b'0.0', b'')],
      dtype=[('x', '<f4'), ('y', 'S3'), ('z', 'O')])
```

## Bài tập liên quan đến mảng con

Khi gán cho các trường là mảng con, giá trị được gán trước tiên sẽ được phát theo hình dạng của mảng con.

## Lập chỉ mục mảng có cấu trúc

## Truy cập các trường riêng lẻ

Các trường riêng lẻ của một mảng có cấu trúc có thể được truy cập và sửa đổi bằng cách lập chỉ mục cho mảng bằng tên trường.

```
>>> ----- x['foo'] mảng([1, 3]) >>> x['foo'] = 10

>>> x
array([(10, 2.), (10, 4.)],
      dtype=[('foo', '<i8'), ('bar', '<f4')])
```

Mảng kết quả là một cái nhìn vào mảng ban đầu. Nó chia sẻ cùng một vị trí bộ nhớ và việc ghi vào khung nhín sẽ sửa đổi mảng ban đầu.

```
>>> y = x['bar'] >>>
y[:] = 11
>>> x
mảng([(10, 11.), (10, 11.)],
      dtype=[('foo', '<i8'), ('bar', '<f4')])
```

Chế độ xem này có cùng loại dtype và kích thước mục như trường được lập chỉ mục, do đó, nó thường là một mảng không có cấu trúc, ngoại trừ trường hợp cấu trúc lồng nhau.

```
>>> y.dtype, y.shape, y.strides
(dtype('float32'), (2,), (12,))
```

Nếu trường được truy cập là một mảng con, thì kích thước của mảng con sẽ được thêm vào hình dạng của kết quả:

```
>>> x = np.zeros((2, 2), dtype=[('a', np.int32), ('b', np.float64, (3, 3))]) >>> x[ 'a'].shape (2,
2) >>> x[ 'b'].shape
(2, 2,
3, 3)
```

## Truy cập nhiều trường

Người ta có thể lập chỉ mục và gán cho một mảng có cấu trúc với chỉ mục nhiều trường, trong đó chỉ mục là danh sách các tên trường.

Cảnh báo: Hành vi của chỉ mục nhiều trường đã thay đổi từ Numpy 1.15 thành Numpy 1.16.

Kết quả của việc lập chỉ mục với chỉ mục nhiều trường là chế độ xem vào mảng ban đầu, như sau:

```
>>> a = np.zeros(3, dtype=[('a', 'i4'), ('b', 'i4'), ('c', 'f4')]) >>> a[ [ 'AC']]
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
mảng([(0, 0.), (0, 0.), (0, 0.)],
      dtype={'names':['a','c'], 'formats':['<i4','<f4'], 'offsets':[0,8], 'itemsize' : 12})
```

Việc gán cho khung nhìn sẽ sửa đổi mảng ban đầu. Các trường của chế độ xem sẽ theo thứ tự chúng được lập chỉ mục. Lưu ý rằng không giống như lập chỉ mục một trường, dtype của chế độ xem có cùng kích thước mục với mảng ban đầu và có các trường ở cùng độ lệch như trong mảng ban đầu và các trường không được lập chỉ mục chỉ bị thiếu.

Cảnh báo: Trong Numpy 1.15, việc lập chỉ mục một mảng có chỉ mục nhiều trường trả về bản sao của kết quả ở trên, nhưng với các trường được đóng gói cùng nhau trong bộ nhớ như thể được truyền qua `numpy.lib.recfunctions.repack_fields`.

Hành vi mới của Numpy 1.16 dẫn đến các byte “đệm” bổ sung tại vị trí của các trường không được lập chỉ mục so với 1.15. Bạn sẽ cần cập nhật bất kỳ mã nào tùy thuộc vào dữ liệu có bối cảnh “được đóng gói”. Ví dụ mã như:

```
>>> a[['a', 'c']].view('i8') # Thất bại trong Numpy 1.16 Traceback (cuộc gọi gần
đây nhất gần đây nhất):
      Tệp "<stdin>", dòng 1, trong <module>
ValueError: Khi thay đổi thành dtype nhỏ hơn, kích thước của nó phải là ước số của . size của dtype ban đầu
```

sẽ cần phải được thay đổi. Mã này đã đưa ra FutureWarning kể từ Numpy 1.12 và mã tương tự đã đưa ra FutureWarning kể từ 1.7.

Trong phiên bản 1.16, một số hàm đã được giới thiệu trong mô-đun `numpy.lib.recfunctions` để giúp người dùng giải thích sự thay đổi này. Đây là `numpy.lib.recfunctions.repack_fields`, `numpy.lib.recfunctions.structured_to_unstructured`, `numpy.lib.recfunctions.structured_to_structured`, `numpy.lib.recfunctions.apply_along_fields`, `numpy.lib.recfunctions.sign_fields_by_name` và `numpy.lib.recfunctions`. Phần bắt buộc.

Hàm `numpy.lib.recfunctions.repack_fields` luôn có thể được sử dụng để tái tạo hành vi cũ vì nó sẽ trả về một bản sao đóng gói của mảng có cấu trúc. Ví dụ, đoạn mã trên có thể được thay thế bằng:

```
>>> from numpy.lib.recfunctions import repack_fields >>> repack_fields(a[['a',
'c']]).view('i8') # được hỗ trợ trong mảng 1.16([0, 0, 0])
```

Hơn nữa, một hàm mới `numpy.lib.recfunctions.structured_to_unstructured`, đây là giải pháp thay thế an toàn và hiệu quả hơn cho những người dùng muốn chuyển đổi mảng có cấu trúc thành mảng không có cấu trúc, như cách xem ở trên thường nhằm mục đích thực hiện. Chức năng này cho phép chuyển đổi an toàn sang loại không có cấu trúc có tính đến phần đệm, thường tránh sao chép và cũng tạo các kiểu dữ liệu khi cần, không giống như chế độ xem. Mã như:

```
>>> b = np.zeros(3, dtype=[('x', 'f4'), ('y', 'f4'), ('z', 'f4')]) >>> b[ ['x', 'z']].view('f4') array([0.,
0., 0., 0., 0., 0., 0.]), dtype =
phao32)
```

có thể được thực hiện an toàn hơn bằng cách thay thế bằng:

```
>>> từ numpy.lib.recfunctions nhập có cấu trúc_to_unstructured >>> có cấu trúc_to_unstructured(b[['x',
'z']]) mảng([0, 0, 0])
```

Việc gán cho một mảng có chỉ mục nhiều trường sẽ sửa đổi mảng ban đầu:

```
>>> a[['a', 'c']] = (2, 3)
>>> một
mảng([(2, 0, 3.), (2, 0, 3.), (2, 0, 3.)], dtype=[('a',
'<i4'), ('b', '<i4'), ('c', '<f4')])
```

Điều này tuân theo các quy tắc gán mảng có cấu trúc được mô tả ở trên. Ví dụ: điều này có nghĩa là người ta có thể hoán đổi giá trị của hai trường bằng cách sử dụng các chỉ mục nhiều trường thích hợp:

```
>>> a[['a', 'c']] = a[['c', 'a']]
```

Lập chỉ mục với một số nguyên để có được vô hướng có cấu trúc

Lập chỉ mục một phần tử của mảng có cấu trúc (với chỉ số nguyên) trả về một vô hướng có cấu trúc:

```
>>> x = np.array([(1, 2., 3.)], dtype='i, f, f') >>> vô hướng =
x[0] >>> vô hướng
(1, 2., 3.)
>>> type(scalar)
<class 'numpy.void'>
```

Không giống như các đại lượng vô hướng dạng khôi khác, các đại lượng vô hướng có cấu trúc có thể thay đổi và hoạt động giống như các khung nhìn vào mảng ban đầu, do đó việc sửa đổi đại lượng vô hướng sẽ sửa đổi mảng ban đầu. Vô hướng có cấu trúc cũng hỗ trợ truy cập và gán theo tên trường:

```
>>> ----- s = x[0] >>> s['bar'] = 100
>>> x
array([(1, 100.), (3, 4.)],
      dtype=[('foo', '<i8'), ('bar', '<f4')])
```

Tương tự như bộ dữ liệu, các đại lượng vô hướng có cấu trúc cũng có thể được lập chỉ mục bằng một số nguyên:

```
>>> vô hướng = np.array([(1, 2., 3.)], dtype='i, f, f')[0] >>> vô hướng[0]
1
>>> vô hướng[1] = 4
```

Do đó, các bộ dữ liệu có thể được coi là Python nguyên gốc tương đương với các kiểu có cấu trúc của numpy, giống như các số nguyên python nguyên gốc tương đương với các kiểu số nguyên của numpy. Các đại lượng vô hướng có cấu trúc có thể được chuyển đổi thành một bộ bằng cách gọi ndarray.item:

```
>>> vô hướng.item(), type(scalar.item()) ((1, 4.0,
3.0), <class 'tuple'>)
```

Xem các mảng có cấu trúc chứa các đối tượng

Để ngăn chặn việc ghi đè các con trỏ đối tượng trong các trường thuộc loại `numpy.object`, `numpy` hiện không cho phép xem các mảng có cấu trúc chứa các đối tượng.

So sánh cấu trúc

Nếu `dtypes` của hai mảng có cấu trúc void bằng nhau, việc kiểm tra tính bằng nhau của các mảng sẽ dẫn đến một mảng boolean có kích thước của mảng ban đầu, với các phần tử được đặt thành True trong đó tất cả các trường của cấu trúc tương ứng đều bằng nhau. Các kiểu `dtype` có cấu trúc bằng nhau nếu tên trường, kiểu `dtype` và tiêu đề giống nhau, bỏ qua độ cuối và các trường theo thứ tự:

```
>>> a = np.zeros(2, dtype=[('a', 'i4'), ('b', 'i4')]) >>> b =
np.ones(2, dtype=[('a', 'i4'), ('b', 'i4')]) >>> a == b
mảng([Sai, Sai])
```

Hiện tại, nếu `dtype` của hai mảng có cấu trúc void không tương đương thì việc so sánh không thành công, trả về giá trị vô hướng `Sai`. Hành vi này không được dùng nữa kể từ phiên bản `numpy 1.10` và sẽ gây ra lỗi hoặc thực hiện so sánh theo từng phần tử trong tương lai.

Các toán tử `<` và `>` luôn trả về `Sai` khi so sánh các mảng có cấu trúc trống và các phép toán số học và bitwise không được hỗ trợ.

### 3.7.4 Mảng ghi

Là một tiện ích tùy chọn, `numpy` cung cấp một lớp con `ndarray`, `numpy.recarray` và các hàm trợ giúp liên quan trong mô-đun `numpy.rec`, cho phép truy cập vào các trường của mảng có cấu trúc theo thuộc tính thay vì chỉ theo chỉ mục. Mảng bản ghi cũng sử dụng kiểu dữ liệu đặc biệt, `numpy.record`, cho phép truy cập trường theo thuộc tính trên các đại lượng vô hướng có cấu trúc thu được từ mảng.

Cách đơn giản nhất để tạo mảng bản ghi là sử dụng `numpy.rec.array`:

```
>>> recordarr = np.rec.array([(1, 2., 'Xin chào'), (2, 3., "World")], dtype=[('foo',
'i4'),('thanh', 'f4'), ('baz', 'S10')])
>>> recordarr.bar
array([ 2.,  3.], dtype=float32) >>>
recordarr[1:2]
rec.array([(2, 3., b'World')], dtype=[('foo', '<i4'), ('bar', '<f4'), ('baz', 'S10')]) >>> recordarr
[1:2].foo array([ 2.],
dtype=int32) >>>
recordarr.foo[1:2]
array([2], dtype=int32) >>>
recordarr[1].baz b'World'
```

`numpy.rec.array` có thể chuyển đổi nhiều loại đối số thành mảng bản ghi, bao gồm cả mảng có cấu trúc:

```
>>> arr = np.array([(1, 2., 'Xin chào'), (2, 3., "World")],
dtype=[('foo', 'i4'), ('bar', 'f4'), ('baz', 'S10')]) >>>
recordarr = np.rec.array(arr)
```

Mô-đun `numpy.rec` cung cấp một số hàm tiện lợi khác để tạo mảng bản ghi, xem quy trình tạo mảng bản ghi.

Có thể thu được biểu diễn mảng bản ghi của mảng có cấu trúc bằng cách sử dụng chế độ xem thích hợp:

```
>>> arr = np.array([(1, 2., 'Xin chào'), (2, 3., "Thế giới")],  
'           dtype=[('foo', 'i4'),('bar', 'f4'), ('baz', 'a10')]) >>> recordarr  
= arr.view(dtype=np.dtype(( np.record, arr.dtype)),  
'           g0=np.recarray)
```

Để thuận tiện, việc xem một ndarray dưới dạng loại np.recarray sẽ tự động chuyển đổi thành kiểu dữ liệu np.record, do đó, dtype có thể bị loại khỏi chế độ xem:

```
>>> recordarr = arr.view(np.recarray) >>>  
recordarr.dtype  
dtype((numpy.record, [(('foo', '<i4'), ('bar', '<f4'), ('baz', 'S10'))]))
```

Để quay lại một ndarray đơn giản, cả dtype và type phải được đặt lại. Chế độ xem sau đây thực hiện như vậy, có tính đến trường hợp bất thường là bản ghi không phải là loại có cấu trúc:

```
>>> arr2 = recordarr.view(recordarr.dtype.fields hoặc recordarr.dtype, np.ndarray)
```

Các trường mảng bản ghi được truy cập theo chỉ mục hoặc theo thuộc tính sẽ được trả về dưới dạng mảng bản ghi nếu trường có kiểu có cấu trúc nhưng nếu không thì ở dạng ndarray đơn giản.

```
>>> recordarr = np.rec.array([('Xin chào', (1, 2)), ("Thế giới", (3, 4))], dtype=[('foo', 'S6'), ('bar',  
'           [('A', int), ('B', int)])])  
>>> type(recordarr.foo)  
<class 'numpy.ndarray'> >>>  
type(recordarr.bar) <class  
'numpy.recarray'>
```

Lưu ý rằng nếu một trường có cùng tên với thuộc tính ndarray thì thuộc tính ndarray sẽ được ưu tiên. Các trường như vậy sẽ không thể truy cập được theo thuộc tính nhưng vẫn có thể truy cập được bằng chỉ mục.

### 3.7.5 Hàm trợ giúp sắp xếp lại

Tập hợp các tiện ích để thao tác với mảng có cấu trúc.

Hầu hết các chức năng này ban đầu được John Hunter triển khai cho matplotlib. Chúng đã được viết lại và mở rộng để thuận tiện.

```
numpy.lib.recfunctions.append_fields(có số, tên, dữ liệu, dtypes=None, fill_value=-1, usemask=True, asrecarray=False)
```

Thêm các trường mới vào một mảng hiện có.

Tên của các trường được đưa ra cùng với các đối số tên, các giá trị tương ứng với các đối số dữ liệu.

Nếu một trường duy nhất được thêm vào, tên, dữ liệu và kiểu dữ liệu không nhất thiết phải là danh sách mà chỉ là giá trị.

Thông số

base [mảng] Nhập mảng để mở rộng.

tên [chuỗi, trình tự] Chuỗi hoặc chuỗi các chuỗi tương ứng với tên của chuỗi mới  
linh vực.

dữ liệu [mảng hoặc chuỗi các mảng] Mảng hoặc chuỗi các mảng lưu trữ các trường để thêm vào  
cần cù.

dtypes [chuỗi các kiểu dữ liệu, tùy chọn] Kiểu dữ liệu hoặc trình tự các kiểu dữ liệu. Nếu Không, các kiểu dữ  
liệu được ước tính từ dữ liệu.

fill\_value [{float}, tùy chọn] Giá trị diền được sử dụng để đệm dữ liệu bị thiếu trên các mảng ngắn hơn.

`usemask [{False, True}, tùy chọn]` Có trả về mảng bị che hay không.

`asrecarray [{False, True}, tùy chọn]` Có trả về một recarray (MaskedRecords) hay không.

`numpy.lib.recfunctions.apply_along_fields(func, arr)`

Áp dụng hàm 'func' dưới dạng rút gọn trên các trường của mảng có cấu trúc.

Điều này tương tự như `apply_along_axis`, nhưng xử lý các trường của mảng có cấu trúc như một trực phụ. Tất cả các trường lần đầu tiên đều được chuyển sang một loại phổ biến theo quy tắc thăng cấp loại từ `numpy.result_type` được áp dụng cho các loại `dtype` của trường.

Thông số

`func [chức năng]` Chức năng áp dụng trên kích thước "trường". Hàm này phải hỗ trợ đổi số trực, như `np.mean`, `np.sum`, v.v.

`arr [ndarray]` Mảng có cấu trúc để áp dụng func.

Trả lại

`out [ndarray]` Kết quả của thao tác cắt bỏ

Ví dụ

```
>>> from numpy.lib import recfunctions as rfn >>> b =
np.array([(1, 2, 5), (4, 5, 7), (7, 8, 11), (10, 11, 12)], dtype=[('x', 'i4'),
('y', 'f4'), ('z', 'f8')]) >>>
rfn.apply_along_fields(np.mean, b)
array([ 2.66666667, 5.33333333, 8.66666667, 11. >>>
rfn.apply_along_fields(np.mean, b[['x', 'z']]) array([ 3.
11. ]], 5.5, 9. ,
```

`numpy.lib.recfunctions.sign_fields_by_name(dst, src, zero_unsigned=True)`

Gán các giá trị từ mảng có cấu trúc này sang mảng có cấu trúc khác theo tên trường.

Thông thường trong `numpy >= 1.14`, việc gán một mảng có cấu trúc cho một trường sao chép khác "theo vị trí", nghĩa là trường đầu tiên từ `src` được sao chép sang trường đầu tiên của `dst`, v.v., bất kể tên trường.

Thay vào đó, hàm này sao chép "theo tên trường", sao cho các trường trong `dst` được gán từ trường có tên giống hệt trong `src`. Điều này áp dụng đệ quy cho các cấu trúc lồng nhau. Đây là cách hoạt động của việc gán cấu trúc trong `numpy >= 1.6` đến `<= 1.13`.

Thông số

`dst [ndarray]`

`src [ndarray]` Mảng nguồn và mảng đích trong quá trình gán.

`zero_unsigned [bool, tùy chọn]` Nếu đúng, các trường trong `dst` không có trường khớp trong `src` sẽ được điền giá trị 0 (không). Đây là hành vi của `numpy <= 1.13`. Nếu sai, các trường đó không được sửa đổi.

`numpy.lib.recfunctions.drop_fields(base, drop_names, usemask=True, asrecarray=False)`

Trả về một mảng mới với các trường trong `drop_names` bị loại bỏ.

Các trường lồng nhau được hỗ trợ.

`..versionchanged: 1.18.0` `drop_fields` trả về một mảng có 0 trường nếu tất cả các trường bị loại bỏ, thay vì trả về `None` như trước đây.

Thông số

cơ sở [mảng] Mảng đầu vào  
**drop\_names** [chuỗi hoặc chuỗi] Chuỗi hoặc chuỗi các chuỗi tương ứng với tên của các cành đồng để thà.  
**usemask** [{False, True}, tùy chọn] Có trả về mảng bị che hay không.  
**asrecarray** [chuỗi hoặc chuỗi, tùy chọn] Trả về một recarray hay mrecarray (asrecarray=True) hay một mảng ndarray đơn giản hoặc mảng bị che với dtype linh hoạt. Mặc định này sai.

## Ví dụ

```
>>> from numpy.lib import recfunctions as rfn >>> a =
np.array([(1, (2, 3.0)), (4, (5, 6.0))], ... dtype=[('a',
np.int64), ('b', [('ba', np.double), ('bb', np.int64)])]) >>> rfn.drop_fields(a, 'a') mảng([(2.,
3.,), (5., 6.,)])  

dtype=[('b', [('ba', '<f8'), ('bb', '<i8')])])  

>>> rfn.drop_fields(a, 'ba')
array([(1, (3.,)), (4, (6.,))], dtype=[('a', '<i8'), ('b', [('bb', '<i8')])]) >>>
rfn.drop_fields(a, ['ba', 'bb']) array([(1,,
(4,) ), dtype=[('a', '<i8')])
```

**numpy.lib.recfunctions.find\_duplicates(a, key=None, ignoremask=False, return\_index=False)**  
Tìm các bản sao trong một mảng có cấu trúc dọc theo một khóa nhất định

Thông số

một mảng đầu vào [giống mảng]

**key** [{string, None}, tùy chọn] Tên của các trường cần kiểm tra các trường trùng lặp. Nếu không có, việc tìm kiếm được thực hiện bởi các bản ghi

**ignoremask** [{True, False}, tùy chọn] Liệu dữ liệu bị che nên được loại bỏ hay xem xét như những bản sao.

**return\_index** [{False, True}, tùy chọn] Có trả về chỉ mục của các giá trị trùng lặp hay không.

## Ví dụ

```
>>> from numpy.lib import recfunctions as rfn >>>
ndtype = [('a', int)] >>> a =
np.ma.array([1, 1, 1, 2, 2, 3, 3], mask=[0, 0,
1, 0, 0, 0, 1]).view(ndtype)
>>> rfn.find_duplicates(a, ignoremask=True, return_index=True)
(masked_array(data=[(1,), (1,), (2,), (2,)],
mask=[(Sai,), (Sai,), (Sai,), (Sai,)],
fill_value=(999999,),  

dtype=[('a', '<i8')]), array([0, 1, 3, 4]))
```

**numpy.lib.recfunctions.flatten\_descr(ndtype)**  
Làm phẳng mô tả kiểu dữ liệu có cấu trúc.

## Ví dụ

```
>>> from numpy.lib import recfunctions as rfn >>>
ndtype = np.dtype([('a', '<i4'), ('b', [('ba', '<f8'), ('bb', '<i4')])]) >>>
rfn.flatten_descr(ndtype) (('a',
dtype('int32')), ('ba', dtype('float64')), ('bb', dtype('int32'))))
```

`numpy.lib.recfunctions.get_fieldstructure(adtype, họ=Không, cha mẹ=Không)`

Trả về một từ điển có các trường lập chỉ mục danh sách các trường mẹ của chúng.

Hàm này được sử dụng để đơn giản hóa việc truy cập vào các trường được lồng trong các trường khác.

Thông số

`adtype [np.dtype]` Kiểu dữ liệu đầu vào

`họ [tùy chọn]` Tên trường được xử lý lần cuối (được sử dụng nội bộ trong quá trình đệ quy).

`cha mẹ [từ điển]` Từ điển của các trường cha mẹ (được sử dụng xen kẽ trong quá trình đệ quy).

## Ví dụ

```
>>> from numpy.lib import recfunctions as rfn >>>
ndtype = np.dtype([('A', int),
                   ('B', [('BA', int),
                          ('BB', [('BBA', int), ('BBB', int)])])])
>>> rfn.get_fieldstructure(ndtype)
... # XXX: có thể hỏi quy, thứ tự BBA và BBB bị hoán đổi {'A': [], 'B': [], 'BA': ['B'], 'BB':
['B'] , 'BBA': ['B', 'BB'], 'BBB': ['B', 'BB
']]}
```

`numpy.lib.recfunctions.get_names(adtype)`

Trả về tên trường của kiểu dữ liệu đầu vào dưới dạng một bộ dữ liệu.

Thông số

`adtype [dtype]` Kiểu dữ liệu đầu vào

## Ví dụ

```
>>> từ quá trình nhập numpy.lib hoạt động trở lại dưới
dạng rfn >>> rfn.get_names(np.empty((1,), dtype=int))
Traceback (cuộc gọi gần đây nhất):
...
AttributionError: Đối tượng 'numpy.ndarray' không có thuộc tính 'name'
```

```
>>> rfn.get_names(np.empty((1,), dtype=[('A',int), ('B', float)]))
Traceback (cuộc gọi gần đây nhất):
...
```

```
AttributionError: Đối tượng 'numpy.ndarray' không có thuộc tính 'names' >>> adtype =
np.dtype([('a', int), ('b', [('ba', int), ('bb', int)]))] >>> rfn.get_names(adtype) ('a', ('b', ('ba',
'bb'))))
```

`numpy.lib.recfunctions.get_names_flat(adtype)`

Trả về tên trường của kiểu dữ liệu đầu vào dưới dạng một bộ dữ liệu. Các cấu trúc lồng nhau được làm phẳng trước.

Thông số

dtype [dtype] Kiểu dữ liệu đầu vào

Ví dụ

```
>>> from numpy.lib import recfunctions as rfn >>>
rfn.get_names_flat(np.empty((1,), dtype=int)) is None Traceback (cuộc gọi gần đây nhất)
gần đây nhất):

AttributionError: Đối tượng 'numpy.ndarray' không có thuộc tính 'names' >>>
rfn.get_names_flat(np.empty((1,), dtype=[('A', int), ('B', float)])) )
Traceback (cuộc gọi gần đây nhất):

AttributionError: Đối tượng 'numpy.ndarray' không có thuộc tính 'names' >>> dtype =
np.dtype([('a', int), ('b', [(('ba', int), ('bb', int))])]) >>> rfn.get_names_flat(dtype) ('a', 'b', 'ba',
'bb')
```

`numpy.lib.recfunctions.join_by(key, r1, r2, jointype='inner', r1postfix='1', r2postfix='2', de-faults=None,
usemask=True, asrecarray=False)`

Nối mảng `r1` và `r2` trên phím khóa.

Khóa phải là một chuỗi hoặc một chuỗi các chuỗi tương ứng với các trường được sử dụng để nối mảng. Một ngoại lệ được đưa ra nếu không thể tìm thấy trường khóa trong hai mảng đầu vào. Cả `r1` và `r2` đều không được có bất kỳ bản sao nào dọc theo khóa: sự hiện diện của các bản sao sẽ khiến kết quả đầu ra không đáng tin cậy. Lưu ý rằng thuật toán không tìm kiếm các bản sao.

Thông số

key [{string, Sequence}] Một chuỗi hoặc một chuỗi các chuỗi tương ứng với các trường được sử dụng cho so sánh.

`r1, r2` [mảng] Mảng có cấu trúc.

jointype [{'inner', 'outer', 'leftouter'}, tùy chọn] Nếu 'inner', trả về các phần tử chung cho cả `r1` và `r2`. Nếu 'outer', trả về các phần tử chung cũng như các phần tử của `r1` không thuộc `r2` và các phần tử không thuộc `r2`. Nếu 'leftouter', trả về các phần tử chung và các phần tử của `r1` không có trong `r2`.

`r1postfix` [chuỗi, tùy chọn] Chuỗi được thêm vào tên của các trường của `r1` có trong `r2` nhưng không có chia khóa.

`r2postfix` [chuỗi, tùy chọn] Chuỗi được thêm vào tên của các trường của `r2` có trong `r1` nhưng không có chia khóa.

mặc định [{dictionary}, tùy chọn] Tên trường ánh xạ từ điển sang giá trị mặc định tương ứng các giá trị.

usemask [{True, False}, tùy chọn] Có trả về MaskedArray hay không (hoặc MaskedRecords là `asrecarray==True`) hoặc `ndarray`.

`asrecarray` [{False, True}, tùy chọn] Trả về một recarray (hoặc MaskedRecords nếu `usemask==True`) hay chỉ là một `ndarray` kiểu linh hoạt.

### Ghi chú

- Đầu ra được sắp xếp theo phím.
- Một mảng tạm thời được hình thành bằng cách loại bỏ các trường không có trong khóa của hai mảng và nối kết quả. Mảng này sau đó được sắp xếp và các mục chung được chọn. Đầu ra được xây dựng bằng cách điền vào các trường với các mục đã chọn. Sự trùng khớp không được giữ nguyên nếu có một số bản sao. , ,

```
numpy.lib.recfunctions.merge_arrays(seqarrays, fill_value=-1, Flatten=False, usemask=False, asrecarray=False)
```

Hợp nhất các trường mảng theo trường.

#### Thông số

**seqarrays** [chuỗi của ndarrays] Chuỗi của mảng  
**fill\_value** [{float}, tùy chọn] Giá trị điền được sử dụng để đệm dữ liệu bị thiếu trên các mảng ngắn hơn.  
**làm phẳng** [{False, True}, tùy chọn] Có thu gọn các trường lồng nhau hay không.  
**usemask** [{False, True}, tùy chọn] Có trả về mảng bị che hay không.  
**asrecarray** [{False, True}, tùy chọn] Có trả về một recarray (MaskedRecords) hay không.

### Ghi chú

- Nếu không có mặt nạ, giá trị còn thiếu sẽ được lấp đầy bằng thứ gì đó, tùy thuộc vào giá trị tương ứng của nó kiểu:
  - -1 cho số nguyên
  - -1,0 đối với số dấu phẩy động
  - '-' cho các ký tự
  - '-1' cho chuỗi
  - Đúng với các giá trị boolean
- XXX: Tôi vừa lấy được những giá trị này theo kinh nghiệm

### Ví dụ

```
>>> from numpy.lib import recfunctions as rfn >>>
rfn.merge_arrays((np.array([1, 2]), np.array([10., 20., 30.]))) array([(1, 10.),
(2, 20.), (-1, 30.)],  

dtype=[('f0', '<i8'), ('f1', '<f8')])
```

```
>>> rfn.merge_arrays((np.array([1, 2], dtype=np.int64),
'np.array([10., 20., 30.]), usemask=False)
mảng([(1, 10.0), (2, 20.0), (-1, 30.0)],
dtype=[('f0', '<i8'), ('f1', '<f8')])
>>> rfn.merge_arrays((np.array([1, 2]).view([('a', np.int64)]), np.array([10.,
20., 30.]), usemask=False,
asrecarray=True) rec.array([(1, 10.),
(2, 20.), (-1, 30.)],
dtype=[('a', '<i8'), ('f1', '<f8')])
```

---

```
numpy.lib.recfunctions.rec_append_fields(cơ sở, tên, dữ liệu, dtypes=Không)
```

Thêm các trường mới vào một mảng hiện có.

Tên của các trường được đưa ra cùng với các đối số tên, các giá trị tương ứng với các đối số dữ liệu.

Nếu một trường duy nhất được thêm vào, tên, dữ liệu và kiểu dữ liệu không nhất thiết phải là danh sách mà chỉ là giá trị.

Thông số

base [mảng] Nhập mảng để mở rộng.

tên [chuỗi, trình tự] Chuỗi hoặc chuỗi các chuỗi tương ứng với tên của chuỗi mới  
lĩnh vực.

dữ liệu [mảng hoặc chuỗi các mảng] Mảng hoặc chuỗi các mảng lưu trữ các trường để thêm vào  
căn cứ.

dtypes [chuỗi các kiểu dữ liệu, tùy chọn] Kiểu dữ liệu hoặc trình tự các kiểu dữ liệu. Nếu Không, các  
kiểu dữ liệu được ước tính từ dữ liệu.

Trả lại

đã thêm\_array [np.recarray]

Xem thêm:

[chấp thêm\\_fields](#)

```
numpy.lib.recfunctions.rec_drop_fields(base, drop_names)
```

Trả về một numpy.recarray mới với các trường trong drop\_names bị loại bỏ.

```
numpy.lib.recfunctions.rec_join(key, r1, r2, jointype='inner', r1postfix='1', r2postfix='2', de-faults=Không)
```

Nối mảng r1 và r2 trên các phím. Thay thế cho join\_by, luôn trả về np.recarray.

Xem thêm:

[hàm tương đương join\\_by](#)

```
numpy.lib.recfunctions.recursive_fill_fields (đầu vào, đầu ra)
```

Điền vào các trường từ đầu ra bằng các trường từ đầu vào, với sự hỗ trợ cho các cấu trúc lồng nhau.

Thông số

đầu vào [ndarray] Mảng đầu vào.

đầu ra [ndarray] Mảng đầu ra.

Ghi chú

- đầu ra ít nhất phải có cùng kích thước với đầu vào

## Ví dụ

```
>>> from numpy.lib import recfunctions as rfn >>> a = np.array([(1,
10.), (2, 20.)], dtype=[('A', np.int64), ('B', np.float64)]) >>> b = np.zeros(3,), dtype=a.dtype) >>> rfn.recursive_fill_fields(a,
b) array([(1, 10.), (2, 20.), (0, 0.)], dtype=[('A',
'<i8'), ('B', '<f8')])
```

`numpy.lib.recfunctions.rename_fields(base, namemapper)`

Đổi tên các trường từ `ndarray` hoặc `recarray` kiểu dữ liệu linh hoạt.

Các trường lồng nhau được hỗ trợ.

Thông số

`base [ndarray]` Mảng đầu vào có các trường phải được sửa đổi.

`namemapper [dictionary]` Từ điển ánh xạ tên trường cũ sang phiên bản mới của chúng.

## Ví dụ

```
>>> from numpy.lib import recfunctions as rfn >>> a = np.array([(1, (2,
[3.0, 30.])), (4, (5, [6.0, 60.])), ...], dtype=[('a', int), ('b', [('ba', float), ('bb', (float,
2))])]) >>> rfn.rename_fields(a, {'a': 'A', 'bb': 'BB'}) array([(1, (2., [3., 30.])), (4, (5., [6.,
60.])), ...], dtype=[('A', '<i8'), ('b', [('ba', '<f8'), ('BB', '<f8',
(2,))])])
```

`numpy.lib.recfunctions.repack_fields(a, căn chỉnh=False, recurse=False)`

Đóng gói lại các trường của mảng có cấu trúc hoặc `dtype` trong bộ nhớ.

Bó cục bộ nhớ của các kiểu dữ liệu có cấu trúc cho phép các trường ở độ lệch byte tùy ý. Điều này có nghĩa là các trường có thể được phân tách bằng các byte đệm, độ lệch của chúng có thể tăng không đơn điệu và chúng có thể chồng lên nhau.

Phương thức này loại bỏ mọi sự trùng lặp và sắp xếp lại các trường trong bộ nhớ để chúng có độ lệch byte tăng dần, đồng thời thêm hoặc xóa các byte đệm tùy thuộc vào tùy chọn `căn chỉnh`, hoạt động giống như tùy chọn `căn chỉnh` cho `np.dtype`.

Nếu `căn chỉnh=False`, phương thức này tạo ra bó cục bộ nhớ "đóng gói" trong đó mỗi trường bắt đầu tại byte mà trường trước đó đã kết thúc và mọi byte đệm đều bị xóa.

Nếu `căn chỉnh=True`, phương thức này tạo ra bó cục bộ nhớ "căn chỉnh" trong đó độ lệch của mỗi trường là bội số của `căn chỉnh` và tổng kích thước mục là bội số của `căn chỉnh` lớn nhất, bằng cách thêm byte đệm nếu cần.

Thông số

một mảng hoặc `dtype [ndarray hoặc dtype]` để đóng gói lại các trường.

`căn chỉnh [boolean]` Nếu đúng, hãy sử dụng bó cục bộ nhớ "căn chỉnh", nếu không thì sử dụng bó cục "đóng gói".

`recurse [boolean]` Nếu đúng, cũng đóng gói lại các cấu trúc lồng nhau.

Trả lại

được đóng gói lại [`ndarray hoặc dtype`] Bản sao của `a` với các trường được đóng gói lại hoặc chính nó nếu không được đóng gói lại cẩn thận.

## Ví dụ

```
>>> từ quá trình nhập numpy.lib hoạt động lại dưới dạng
rfn >>> def print_offsets(d):
...     print("offsets:", [d.fields[name][1] cho tên trong d.names])
...     print("itemsize:", d.itemsize)
...
>>> dt = np.dtype('u1, <i8, <f8', align=True) >>> dt
dtype({'names':['f0','f1','f2'], 'formats':['u1','<i8','<f8'], 'offsets':[0,8, 16], 'itemsize':24},align=True) >>>
print_offsets(dt) offsets:
[0, 8, 16] itemsize: 24

>>> đóng gói_dt = rfn.repack_fields(dt) >>>
đóng gói_dt
dtype([('f0', 'u1'), ('f1', '<i8'), ('f2', '<f8')]) >>>
print_offsets(packed_dt) offset:
[0, 1, 9] kích thước
mục: 17
```

`numpy.lib.recfunctions.require_fields(mảng, require_dtype)`

Truyền một mảng có cấu trúc sang một dtype mới bằng cách gán theo tên trường.

Hàm này gán tên từ mảng cũ sang mảng mới nên giá trị của một trường trong mảng đầu ra là giá trị của trường có cùng tên trong mảng nguồn. Điều này có tác dụng tạo ra một ndarray mới chỉ chứa các trường “bắt buộc” theo `require_dtype`.

Nếu tên trường trong `require_dtype` không tồn tại trong mảng đầu vào, trường đó sẽ được tạo và đặt thành 0 trong mảng đầu ra.

## Thông số

một mảng [ndarray] để truyền  
kiểu dữ liệu bắt buộc `dtype` [dtype] cho mảng đầu ra

Trả lại

out mảng [ndarray] với dtype mới, với các giá trị trường được sao chép từ các trường trong mảng đầu vào có cùng tên

## Ví dụ

```
>>> from numpy.lib import recfunctions as rfn >>> a =
np.ones(4, dtype=[('a', 'i4'), ('b', 'f8'), ('c', 'u1')]) >>>
rfn.require_fields(a, [('b', 'f4'), ('c', 'u1')]) array([(1.,
1), (1., 1), (1., 1)], dtype=[('b', '<f4'),
('c', 'u1')])
>>> rfn.require_fields(a, [('b', 'f4'), ('newf', 'u1')]) array([(1.,
0), (1., 0), (1., 0)], dtype=[('b', '<f4'),
('newf', 'u1')])
```

`numpy.lib.recfunctions.stack_arrays(arrays, defaults=None, usemask=True, asrecarray=False, autoconvert=False)`

Xếp chồng các trường mảng theo trường

## Thông số

mảng [mảng hoặc chuỗi] Trình tự các mảng đầu vào.

mặc định [từ điển, tùy chọn] Tên trường ánh xạ từ điển sang mặc định tương ứng các giá trị.

usemask [{True, False}, tùy chọn] Có trả về MaskedArray hay không (hoặc MaskedRecords là asrecarray==True) hoặc ndarray.

asrecarray [{False, True}, tùy chọn] Trả về một recarray (hoặc MaskedRecords nếu use-mask==True) hay chỉ là một ndarray kiểu linh hoạt.

tự động chuyển đổi [{False, True}, tùy chọn] Liệu có tự động chuyển loại trường sang tối đa.

Ví dụ

```
>>> from numpy.lib import recfunctions as rfn >>> x =
np.array([1, 2,]) >>
rfn.stack_arrays(x) is x True

>>> z = np.array([('A', 1), ('B', 2)], dtype=[('A', '|S3'), ('B', float)]) >>> zz = np.array([('a', 10., 100.), ('b',
20., 200.), ('c', 30., 300.)], ... dtype=[('A', '|S3'), ('B', np.double), ('C', np.double)]) >>> test =
rfn.stack_arrays((z,zz)) >>> kiểm tra

Masked_array(data=[(b'A', 1.0, --), (b'B', 2.0, --), (b'a', 10.0, 100.0),
(b'b', 20.0, 200.0), (b'c', 30.0, 300.0)],
mật na=[(Sai, Sai, Đúng), (Sai, Sai, Đúng), (Sai, Sai, Sai), (Sai,
Sai, Sai), (Sai, Sai, Sai)], fill_value=(b'N/A', 1.e+20,
1.e+20),
dtype=[('A', 'S3'), ('B', '<f8'), ('C', '<f8')])
```

`numpy.lib.recfunctions.structured_to_unstructured(arr, dtype=None, copy=False, cast='unsafe')`

Chuyển đổi và mảng có cấu trúc nD thành mảng không có cấu trúc (n+1)-D.

Mảng mới sẽ có chiều cuối cùng mới có kích thước bằng với số phần tử trường của mảng đầu vào.

Nếu không được cung cấp, kiểu dữ liệu đầu ra sẽ được xác định từ quy tắc thăng hạng kiểu gọn gàng áp dụng cho tất cả các kiểu dữ liệu trường.

Các trường lồng nhau, cũng như từng phần tử của bất kỳ trường mảng con nào, đều được tính là một phần tử trường đơn lẻ.

Thông số

`arr [ndarray]` Mảng có cấu trúc hoặc dtype để chuyển đổi. Không thể chứa kiểu dữ liệu đối tượng.

`dtype [dtype, tùy chọn]` Dtype của mảng không có cấu trúc đầu ra.

`sao chép [bool, tùy chọn]` Xem đối số sao chép vào ndarray.astype. Nếu đúng, luôn trả lại một bản sao.

Nếu sai và các yêu cầu về dtype được đáp ứng, một khung nhìn sẽ được trả về.

`truyền [{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, tùy chọn]` Xem đối số truyền của ndarray.astype.

Kiểm soát loại truyền dữ liệu nào có thể xảy ra.

Trả lại

Không có cấu trúc [ndarray] Mảng không có cấu trúc có thêm một chiều.

Ví dụ

```
>>> from numpy.lib import recfunctions as rfn >>> a =
np.zeros(4, dtype=[('a', 'i4'), ('b', 'f4,u2'), ('c ', 'f4', 2)])
>>> một
mảng([(0, (0., 0), [0., 0.]), (0, (0., 0), [0., 0.]),
       (0, (0., 0), [0., 0.]), (0, (0., 0), [0., 0.])],
      dtype=[('a', '<i4'), ('b', [('f4', '<f4'), ('f1', '<u2')]), ('c ', '<f4', (2, ))]) >>>

rfn.structured_to_unstructured(a) mảng([[0., 0.,
0., 0., 0., 0., 0., 0.],
[0., 0., 0., 0., 0., 0., 0., 0.]])
```

```
>>> b = np.array([(1, 2, 5), (4, 5, 7), (7, 8 ,11), (10, 11, 12)], dtype=[('x' , 'i4'),
. ('y', 'f4'), ('z', 'f8')]) >>>
np.mean(rfn.structured_to_unstructured(b[['x', 'z']]), axis=-1) mảng([ 3. , 5.5, 9.
, 11.])
```

```
numpy.lib.recfunctions.unstructured_to_structured(arr, dtype=None, name=None, align=False, copy=False, casting='unsafe')
```

Chuyển đổi và nD mảng không có cấu trúc thành mảng có cấu trúc (n-1)-D.

Chiều cuối cùng của mảng đầu vào được chuyển đổi thành một cấu trúc, với số phần tử trường bằng kích thước chiều cuối cùng của mảng đầu vào. Theo mặc định, tất cả các trường đầu ra đều có dtype của mảng đầu vào, nhưng thay vào đó, một dtype có cấu trúc đầu ra có số lượng phần tử trường bằng nhau có thể được cung cấp.

Các trường lồng nhau, cũng như từng phần tử của bất kỳ trường mảng con nào, đều được tính vào số phần tử trường.

Thông số

`arr [ndarray]` Mảng hoặc dtype không có cấu trúc để chuyển đổi.

`dtype [dtype, tùy chọn]` Dtype có cấu trúc của mảng đầu ra

`tên [danh sách các chuỗi, tùy chọn]` Nếu dtype không được cung cấp, điều này chỉ định tên trường cho dtype đầu ra, theo thứ tự. Trường dtypes sẽ giống với mảng đầu vào.

`căn chỉnh [boolean, tùy chọn]` Có tạo bộ nhớ được căn chỉnh hay không.

`sao chép [bool, tùy chọn]` Xem đối số sao chép vào ndarray.astype. Nếu đúng, luôn trả lại một bản sao.

Nếu sai và các yêu cầu về dtype được đáp ứng, một khung nhìn sẽ được trả về.

`truyền [{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}], tùy chọn]` Xem đối số truyền của ndarray.astype.

Kiểm soát loại truyền dữ liệu nào có thể xảy ra.

Trả lại

`có cấu trúc [ndarray]` Mảng có cấu trúc với ít chiều hơn.

Ví dụ

```
>>> from numpy.lib import recfunctions as rfn >>> dt =
np.dtype([('a', 'i4'), ('b', 'f4,u2'), ('c', 'f4 ', 2)]) >>> a =
np.arange(20).reshape((4,5))
>>> một
mảng([[ 0,  1,  2,  3,  4], [ 5,  6,  7,
     8,  9], [10, 11, 12, 13,
    14], [15, 16, 17, 18, 19] ]]

>>> rfn.unstructured_to_structured(a, dt) mảng([( 0,
( 1., 2), [ 3., 4]), ( 5, ( 6., 7), [ 8., 9.]),
(10, (11., 12), [13., 14.]), (15, (16., 17), [18., 19.])],
dtype=[('a', '<i4'), ('b', [('f0', '<f4'), ('f1', '<u2')]), ('c', ' <f4', (2, ))])
```

### 3.8 Viết vùng chứa mảng tùy chỉnh

Cơ chế điều phối của Numpy, được giới thiệu trong phiên bản numpy v1.16 là phương pháp được đề xuất để viết các vùng chứa mảng N chiều tùy chỉnh tương thích với API numpy và cung cấp cách triển khai tùy chỉnh chức năng numpy. Ứng dụng bao gồm `bàn` mảng, mảng N chiều được phân bố trên nhiều nút và `cupy` mảng, mảng N chiều trên GPU.

Để hiểu cách viết vùng chứa mảng tùy chỉnh, chúng ta sẽ bắt đầu với một ví dụ đơn giản có tiện ích khá hẹp nhưng minh họa các khái niệm liên quan.

```
>>> nhập numpy dưới dạng
np >>> lớp DiagonalArray:
'     def __init__(self, N, value): self._N = N
'     self._i = giá
'     trả def __repr__(self):
'     return f"{self.__class__.__name__}
'             (N={self._N}, value={self. _i})" def __array__(self): trả về self._i * np.eye(self._N)
'
'
'
```

Mảng tùy chỉnh của chúng tôi có thể được khởi tạo như sau:

```
>>> mảng = DiagonalArray(5, 1)
>>> mảng
DiagonalArray(N=5, value=1)
```

Chúng ta có thể chuyển đổi thành một mảng có nhiều mảng bằng cách sử dụng `numpy.array` hoặc `numpy.asarray`, phương thức này sẽ gọi phương thức `__array__` của nó để thu được một `numpy.ndarray` tiêu chuẩn.

```
>>> np.asarray(arr)
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.]])
```

Nếu chúng ta thao tác trên `arr` với hàm numpy, numpy sẽ lại sử dụng giao diện `__array__` để chuyển đổi nó thành một mảng và sau đó áp dụng hàm theo cách thông thường.

```
>>> np.multiply(arr, 2)
mảng([[2., 0., 0., 0.],
       [0., 2., 0., 0., 0.],
       [0., 0., 2., 0., 0.],
       [0., 0., 0., 2., 0.],
       [0., 0., 0., 0., 2.]])
```

Lưu ý rằng kiểu trả về là một `numpy.ndarray` tiêu chuẩn.

```
>>> gõ (mảng)
numpy.ndarray
```

Làm cách nào chúng ta có thể chuyển kiểu `mảng` tùy chỉnh của mình thông qua hàm này? Numpy cho phép một lớp chỉ ra rằng nó muốn xử lý các tính toán theo cách được xác định tùy chỉnh thông qua các tương tác `_array_ufunc_` và `_mảng_hàm_`. Chúng ta hãy thực hiện từng cái một, bắt đầu bằng `_array_ufunc_`. Phương pháp này bao gồm `ufuncs`, một lớp hàm bao gồm, ví dụ: `numpy.multiply` và `numpy.sin`.

`_array_ufunc_` nhận được:

- `ufunc`, một hàm như `numpy.multiply`
- phương thức, một chuỗi, phân biệt giữa `numpy.multiply(...)` và các biến thể như `numpy.multiply`. bên ngoài, `numpy.multiply.accumulate`, v.v. Đối với trường hợp thông thường, `numpy.multiply(...)`, phương thức == '`_gọi_`'.
- đầu vào, có thể là hỗn hợp của nhiều loại khác nhau
- `kwargs`, đối số từ khóa được truyền vào hàm

Trong ví dụ này chúng ta sẽ chỉ xử lý phương thức `_call_`.

```
>>> từ số nhập Số
>>> lớp DiagonalArray:
    def __init__(self, N, value):
        tự._N = N
        self._i = giá trị
    chắc chắn __repr__(tự):
        return f"{self.__class__.__name__}(N={self._N}, value={self._i})"
    def __array__(tự):
        trả về self._i * np.eye(self._N)
    def __array_ufunc__(self, ufunc, phương thức, *đầu vào, **kwargs):
        nếu phương thức == '__call__':
            N = Không có
            vô hướng = []
            cho đầu vào trong đầu vào:
                if isinstance(input, Number):
                    vô hướng.append(đầu vào)
                Elif isinstance(input, self.__class__):
                    vô hướng.append(input._i)
                    nếu N không phải là Không:
                        nếu N != self._N:
                            tăng TypeError("kích thước không nhất quán")
                    nếu không thì:
                        N = tự._N
                nếu không thì:
                    trả về Chưa thực hiện
            return self.__class__(N, ufunc(*scalars, **kwargs))
        nếu không thì:
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
' trả về Chưa thực hiện'
```

Bây giờ kiểu mảng tùy chỉnh của chúng tôi chuyển qua các hàm có nhiều khôi.

```
>>> mảng = DiagonalArray(5, 1)
>>> np.multiply(arr, 3)
DiagonalArray(N=5, value=3)
>>> np.add(arr, 3)
DiagonalArray(N=5, value=4)
>>> np.sin(arr)
DiagonalArray(N=5, value=0.8414709848078965)
```

Tại thời điểm này arr + 3 không hoạt động.

```
>>> mảng + 3
TypeError: (các) loại toán hạng không được hỗ trợ cho *: 'DiagonalArray' và 'int'
```

Để hỗ trợ nó, chúng ta cần xác định các giao diện Python `__add__`, `__lt__`, v.v. để gửi đến giao diện tương ứng ufunc. Chúng ta có thể đạt được điều này một cách thuận tiện bằng cách kế thừa từ mixin `NDArrayOperatorsMixin`.

```
>>> nhập numpy.lib.mixins
>>> lớp DiagonalArray(numpy.lib.mixins.NDArrayOperatorsMixin):
    def __init__(self, N, value):
        tự._N = N
        self._i = giá trị
    đặc chán __repr__(tự):
        return f"{self.__class__.__name__}(N={self._N}, value={self._i})"
    def __array__(tự):
        trả về self._i * np.eye(self._N)
    def __array_ufunc__(self, ufunc, phương thức, *đầu vào, **kwargs):
        nếu phương thức == '__call__':
            N = Không có
            vô hướng = []
            cho đầu vào trong đầu vào:
                if isinstance(input, Number):
                    vô hướng.append(đầu vào)
                Elif isinstance(input, self.__class__):
                    vô hướng.append(input._i)
                    nếu N không phải là Không:
                        nếu N != self._N:
                            tăng TypeError("kích thước không nhất quán")
                    nếu không thì:
                        N = tự._N
                    nếu không thì:
                        trả về Chưa thực hiện
            trả về self.__class__(N, ufunc(*scalars, **kwargs))
        nếu không thì:
            trả về Chưa thực hiện
```

```
>>> mảng = DiagonalArray(5, 1)
>>> mảng + 3
DiagonalArray(N=5, value=4)
>>> mảng > 0
DiagonalArray(N=5, value=True)
```

Bây giờ hãy giải quyết `__array_function__`. Chúng tôi sẽ tạo dict ánh xạ các hàm phức tạp tới các biến thể tùy chỉnh của chúng tôi.

```
>>> HANDLED_FUNCTIONS = {}

>>> lớp DiagonalArray(numpy.lib.mixins.NDArrayOperatorsMixin):
    def __init__(self, N, value):
        tự._N = N
        self._i = giá trị
    đặc chẩn __repr__(tự):
        return f"{self.__class__.__name__}(N={self._N}, value={self._i})"

    def __array__(tự):
        trả về self._i * np.eye(self._N)

    def __array_ufunc__(self, ufunc, phương thức, *đầu vào, **kwargs):
        nếu phương thức == '__call__':
            N = Không có
            vô hướng = []
            cho đầu vào trong đầu vào:
                # Trong trường hợp này, chúng tôi chỉ chấp nhận số vô hướng hoặc DiagonalArrays.
                if isinstance(input, Number):
                    vô hướng.append(đầu vào)
                Elif isinstance(input, self.__class__):
                    vô hướng.append(input._i)
                    nếu N không phải là Không:
                        nếu N != self._N:
                            tăng TypeError("kích thước không nhất quán")
                    nếu không thì:
                        N = tự._N
                nếu không thì:
                    trả về Chưa thực hiện
            return self.__class__(N, ufunc(*scalars, **kwargs))

        nếu không thì:
            trả về Chưa thực hiện
    def __array_function__(self, func, type, args, kwargs):
        nếu chức năng không có trong HANDLED_FUNCTIONS:
            trả về Chưa thực hiện
            # Lưu ý: điều này cho phép các lớp con không ghi đè
            # __array_function__ để xử lý các đối tượng DiagonalArray.
            nếu không phải all(isinstance(t, self.__class__) for t trong các loại):
                trả về Chưa thực hiện
            trả về HANDLED_FUNCTIONS[func](*args, **kwargs)
```

Một mẫu thuận tiện là xác định một công cụ trang trí có thể được sử dụng để thêm các chức năng vào `HANDLED_FUNCTIONS`.

```
>>> dụng cụ def (np_function):
    "Đăng ký triển khai __array_function__ cho đối tượng DiagonalArray."
    trang trí def (func):
        HANDLED_FUNCTIONS[np_function] = chức năng
        hàm trả về
    trở lại trang trí
```

Bây giờ chúng ta viết phần triển khai các hàm gọn gàng cho `DiagonalArray`. Để hoàn thiện, hỗ trợ việc sử dụng `arr.sum()` thêm một phương thức `sum` gọi `numpy.sum(self)` và tương tự cho giá trị trung bình.

```
>>> @implements(np.sum)
... tổng def (arr):
    "Triển khai np.sum cho đối tượng DiagonalArray"
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```

'       trả về mảng._i * mảng._N
'

>>> @implements(np.mean) ...
def Mean(arr):
    "Triển khai np.mean cho đối tượng DiagonalArray" return arr._i / arr._N

'

>>> arr = DiagonalArray(5, 1) >>>
np.sum(arr) 5

>>> np.mean(arr)
0,2

```

Nếu người dùng cố gắng sử dụng bất kỳ hàm numpy nào không có trong HANDLED\_FUNCTIONS, thì TypeError sẽ tăng lên bởi numpy, cho biết thao tác này không được hỗ trợ. Ví dụ: việc ghép hai DiagonalArrays không tạo ra một mảng đường chéo khác, vì vậy nó không được hỗ trợ.

```

>>> np.concatenate([arr, arr])
TypeError: không tìm thấy triển khai nào cho 'numpy.concatenate' trên các loại triển khai __array_function__: [<class
'__main__.DiagonalArray'>]

```

Ngoài ra, việc triển khai tổng và trung bình của chúng tôi không chấp nhận các đối số tùy chọn mà cách triển khai của numpy thực hiện.

```

>>> np.sum(arr, axis=0)
TypeError: sum() nhận được đối số từ khóa không mong muốn 'trục'

```

Người dùng luôn có tùy chọn chuyển đổi thành numpy.ndarray bình thường bằng numpy.asarray và sử dụng numpy tiêu chuẩn từ đó.

```

>>> np.concatenate([np.asarray(arr), np.asarray(arr)]) 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.], [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.]])

```

Tham khảo mã nguồn bàn và mã nguồn cupy để biết thêm các ví dụ hoạt động đầy đủ hơn về vùng chứa mảng tùy chỉnh.

Xem thêm NEP 18.

## 3.9 Phân lớp ndarray

### 3.9.1 Giới thiệu

Phân lớp ndarray tương đối đơn giản nhưng nó có một số vấn đề phức tạp so với các đối tượng Python khác. Trên trang này, chúng tôi giải thích cơ chế cho phép bạn phân lớp ndarray và ý nghĩa của việc triển khai một lớp con.

#### ndarrays và tạo đối tượng

Việc phân lớp ndarray phức tạp vì thực tế là các thể hiện mới của lớp ndarray có thể xuất hiện theo ba cách khác nhau. Đó là:

1. Lệnh gọi hàm tạo rõ ràng - như trong `MySubClass(params)`. Đây là con đường thông thường để tạo phiên bản Python.
2. Xem truyền - truyền một ndarray hiện có dưới dạng một lớp con nhất định
3. Mới từ mẫu - tạo một phiên bản mới từ một phiên bản mẫu. Các ví dụ bao gồm trả về các lát cắt từ một mảng được phân lớp con, tạo các kiểu trả về từ ufuncs và sao chép mảng. Xem [Tạo mới từ mẫu](#) để biết thêm chi tiết

Hai cái cuối cùng là đặc điểm của ndarrays - để hỗ trợ những thứ như cắt mảng. Sự phức tạp của việc phân lớp ndarray là do các cơ chế numpy phải hỗ trợ hai tuyến tạo cá thể sau này.

### 3.9.2 Xem truyền hình

Truyền chế độ xem là cơ chế ndarray tiêu chuẩn mà theo đó bạn lấy một ndarray của bất kỳ lớp con nào và trả về chế độ xem của mảng dưới dạng một lớp con (được chỉ định) khác:

```
>>> import numpy as np >>> # tạo
một lớp con ndarray hoàn toàn vô dụng >>> class C(np.ndarray): pass >>> # tạo
một ndarray tiêu chuẩn >>> arr =
np.zeros((3,) ) >>> # hãy xem nó như lớp con vô
dụng của chúng ta

>>> c_arr = arr.view(C)
>>> gõ(c_arr) <class
'C'>
```

### 3.9.3 Tạo mới từ mẫu

Các phiên bản mới của lớp con ndarray cũng có thể xuất hiện theo một cơ chế rất giống với [việc truyền View](#), khi nhận thấy nó cần tạo một phiên bản mới từ một phiên bản mẫu. Nơi rõ ràng nhất mà điều này xảy ra là khi bạn đang lấy các lát mảng được phân lớp con. Ví dụ:

```
>>> v = c_arr[1:] >>>
type(v) # view thuộc loại 'C' <class 'C'>

>>> v là c_arr # nhưng là phiên bản mới
SAI
```

Lát cắt này là một chế độ xem trên dữ liệu `c_arr` ban đầu. Vì vậy, khi chúng ta xem từ ndarray, chúng ta trả về một ndarray mới, cùng lớp, trả đến dữ liệu trong bản gốc.

Có những điểm khác trong việc sử dụng ndarray mà chúng ta cần các dạng xem như vậy, chẳng hạn như sao chép mảng (`c_arr.copy()`), tạo mảng đầu ra ufunc (xem thêm `__array_wrap__` để biết ufuncs và các hàm khác) và các phương thức rút gọn (như `c_arr.mean()`).

### 3.9.4 Mối quan hệ giữa việc truyền chế độ xem và mẫu mới

Những con đường này đều sử dụng cùng một máy móc. Chúng tôi phân biệt ở đây vì chúng mang lại đầu vào khác nhau cho các phương pháp của bạn. Cụ thể, **Truyền chế độ xem** có nghĩa là bạn đã tạo một phiên bản mới của loại mảng từ bất kỳ lớp con tiềm năng nào của ndarray. **Tạo mới** từ **mẫu** có nghĩa là bạn đã tạo một phiên bản mới của lớp từ một phiên bản có sẵn, ví dụ: cho phép bạn sao chép qua các thuộc tính dành riêng cho lớp con của bạn.

### 3.9.5 Ý nghĩa của việc phân lớp con

Nếu chúng ta phân lớp ndarray, chúng ta không chỉ cần xử lý việc xây dựng rõ ràng kiểu mảng mà còn cả **Xem truyền** hoặc **Tạo mới** từ **mẫu**. NumPy có bộ máy để thực hiện việc này và bộ máy này khiến cho việc phân lớp con hơi không chuẩn.

Có hai khía cạnh đối với bộ máy mà ndarray sử dụng để hỗ trợ các khung nhìn và mẫu mới trong các lớp con.

Đầu tiên là việc sử dụng phương thức `ndarray.__new__` cho công việc chính là khởi tạo đối tượng, thay vì phương thức `__init__` thông thường hơn. Thứ hai là việc sử dụng phương thức `__array_finalize__` để cho phép các lớp con dọn dẹp sau khi tạo các chế độ xem và phiên bản mới từ các mẫu.

Sơ lược về Python ngắn gọn về `__new__` và `__init__`

`__new__` là một phương thức Python tiêu chuẩn và nếu có thì sẽ được gọi trước `__init__` khi chúng ta tạo một phiên bản lớp. Xem tài liệu python `__new__` để biết thêm chi tiết.

Ví dụ: hãy xem xét mã Python sau:

```
lớp C(đối tượng): def
    __new__(cls, *args): print('Cls in
        __new__', cls) print('Args in __new__', args)
        # Phương thức `object` kiểu __new__ nhận một đối số duy nhất. trả về đối tượng.__new__(cls)

    def __init__(self, *args):
        print('type(self) in __init__', type(self)) print('Args in
            __init__', args)
```

có nghĩa là chúng tôi nhận được:

```
>>> c = C('xin chào')
Cls trong __new__: <class 'C'>
Các đối số trong __new__: ('hello',)
gõ(self) trong __init__: <class 'C'>
Args trong __init__: ('xin chào',)
```

Khi chúng ta gọi `C('hello')`, phương thức `__new__` lấy lớp riêng của nó làm đối số đầu tiên và đối số được truyền là chuỗi 'hello'. Sau khi python gọi `__new__`, nó thường (xem bên dưới) gọi phương thức `__init__` của chúng ta, với đầu ra là `__new__` làm đối số đầu tiên (bây giờ là một thể hiện của lớp) và các đối số được truyền theo sau.

Như bạn có thể thấy, đối tượng có thể được khởi tạo trong phương thức `__new__` hoặc phương thức `__init__` hoặc cả hai, và trên thực tế, ndarray không có phương thức `__init__`, vì tất cả việc khởi tạo đều được thực hiện trong phương thức `__new__`.

Tại sao nên sử dụng `__new__` thay vì chỉ `__init__` thông thường? Bởi vì trong một số trường hợp, như đối với `ndarray`, chúng ta muốn có thể trả về một đối tượng của một lớp khác. Hãy xem xét những điều sau:

```
lớp D (C):
    định nghĩa __new__(cls, *args):
        print('D cls is:', cls) print('D
        args in __new__: ', args) return C.__new__(C, *args)

    def __init__(self, *args): # chúng tôi
        không bao giờ đến được đây
        print('In D __init__')
```

điều đó có nghĩa là:

```
>>> obj = D('xin chào')
D cls là: <class 'D'>
D lập luận trong __new__: ("xin chào",)
Cls trong __new__: <class 'C'>
Các đối số trong __new__: ('hello',) >>
type(obj) <class 'C'>
```

Định nghĩa của `C` vẫn giống như trước, nhưng đối với `D`, phương thức `__new__` trả về một thể hiện của lớp `C` chứ không phải `D`. Lưu ý rằng phương thức `__init__` của `D` không được gọi. Nói chung, khi phương thức `__new__` trả về một đối tượng của lớp khác với lớp mà nó được định nghĩa, thì phương thức `__init__` của lớp đó sẽ không được gọi.

Đây là cách các lớp con của lớp `ndarray` có thể trả về các khung nhìn duy trì kiểu lớp. Khi xem, máy móc `ndarray` tiêu chuẩn sẽ tạo đối tượng `ndarray` mới với nội dung như:

```
obj = ndarray.__new__(kiểu con, hình dạng, ...)
```

trong đó `subdtype` là lớp con. Do đó, khung nhìn được trả về có cùng lớp với lớp con, thay vì thuộc lớp `ndarray`.

Điều đó giải quyết được vấn đề trả về các chế độ xem cùng loại, nhưng bây giờ chúng ta gặp phải một vấn đề mới. Bộ máy của `ndarray` có thể thiết lập lớp theo cách này, theo các phương thức tiêu chuẩn của nó để lấy các khung nhìn, nhưng phương thức `ndarray __new__` không biết gì về những gì chúng ta đã làm trong phương thức `__new__` của riêng mình để đặt các thuộc tính, v.v. (Bên cạnh đó - tại sao không gọi `obj = subdtype.__new__(... then?` Bởi vì chúng ta có thể không có phương thức `__new__` có cùng chữ ký cuộc gọi).

Vai trò của `__array_finalize__`

`__array_finalize__` là cơ chế mà numpy cung cấp để cho phép các lớp con xử lý các cách khác nhau mà các phiên bản mới được tạo.

Hãy nhớ rằng các thể hiện của lớp con có thể xảy ra theo ba cách sau:

1. Lệnh gọi hàm tạo rõ ràng (`obj = MySubClass(params)`). Điều này sẽ gọi chuỗi thông thường của `MySubClass.__new__` sau đó (nếu nó tồn tại) `MySubClass.__init__`.
2. Xem buổi casting
3. Tạo mới từ mẫu

Phương thức `MySubClass.__new__` của chúng tôi chỉ được gọi trong trường hợp lệnh gọi hàm tạo rõ ràng, vì vậy chúng tôi không thể dựa vào `MySubClass.__new__` hoặc `MySubClass.__init__` để xử lý việc truyền chế độ xem và mẫu mới từ mẫu.

Hóa ra `MySubClass.__array_finalize__` được gọi cho cả ba phương pháp tạo đối tượng, vì vậy đây là nơi công việc quản lý việc tạo đối tượng của chúng ta thường diễn ra.

- Đối với lệnh gọi hàm tạo rõ ràng, lớp con của chúng ta sẽ cần tạo một thể hiện ndarray mới của lớp riêng của nó. Trong thực tế, điều này có nghĩa là chúng ta, những tác giả của mã, sẽ cần thực hiện lệnh gọi tới `ndarray.__new__(MySubClass,...)`, một lệnh gọi được chuẩn bị theo phân cấp lớp tới `super(MySubClass, cls).__new__(cls, ...)`, hoặc xem việc truyền một mảng hiện có (xem bên dưới)
- Để truyền chế độ xem và mẫu mới từ mẫu, tương đương với `ndarray.__new__(MySubClass,...)` là được gọi là ở cấp độ C.

Các đối số mà `__array_finalize__` nhận được sẽ khác nhau đối với ba phương thức tạo phiên bản ở trên.

Đoạn mã sau cho phép chúng ta xem trình tự cuộc gọi và đối số:

```
nhập numpy dưới dạng np

lớp C (np.ndarray):
    def __new__(cls, *args, **kwargs):
        print('In __new__ with class %s' % cls) return super(C,
        cls).__new__(cls, *args, **kwargs)

    def __init__(self, *args, **kwargs):
        # trong thực tế có thể bạn sẽ không cần hoặc không muốn có một phương thức __init__ # cho lớp
        # con của bạn print('In __init__ with
        class %s' % self.__class__)

    def __array_finalize__(self, obj): print('In
        array_finalize:') print(' self type is
        %s' % type(self)) print(' obj type is %s' % type(obj))
```

Hiện nay:

```
>>> # Hàm tạo rõ ràng >>> c = C((10,))

Trong __new__ với lớp <class 'C'> Trong
array_finalize: loại tự là
<class 'C'> loại obj là <type
'NoneType'> Trong __init__ với lớp <class
'C'> >>> # Xem truyền >>> a = np.arange(10) >>
cast_a = a.view(C)

Trong array_finalize: self
    type là <class 'C'> obj type is <type
    'numpy.ndarray'> >>> # Slicing (ví dụ về new-from-
template) >>> cv = c[:1]

Trong array_finalize: loại
    tự là <class 'C'> loại obj là <class
    'C'>
```

Chữ ký của `__array_finalize__` là:

```
def __array_finalize__(self, obj):
```

Người ta thấy rằng lệnh gọi siêu, đi đến `ndarray.__new__`, chuyển `__array_finalize__` đổi tương mới, của lớp riêng của chúng ta (`self`) cũng như đổi tương mà chế độ xem đã được thực hiện (`obj`). Như bạn có thể thấy từ kết quả đầu ra ở trên, `self` luôn là một phiên bản mới được tạo của lớp con của chúng ta và loại `obj` khác nhau đối với ba phương thức tạo phiên bản:

- Khi được gọi từ hàm tạo rõ ràng, `obj` là Không có

- Khi được gọi từ chế độ truyền khung nhìn, obj có thể là một thực thể của bất kỳ lớp con nào của ndarray, bao gồm cả lớp con của chúng ta.
- Khi được gọi trong new-from-template, obj là một phiên bản khác của lớp con của chính chúng ta mà chúng ta có thể sử dụng để cập nhật. trường hợp tự mới.

Vì `__array_finalize__` là phương thức duy nhất luôn thấy các phiên bản mới được tạo nên đây là nơi hợp lý để điều các giá trị mặc định của phiên bản cho các thuộc tính đối tượng mới, cùng với các tác vụ khác.

Điều này có thể rõ ràng hơn với một ví dụ.

### 3.9.6 Ví dụ đơn giản - thêm thuộc tính bổ sung vào ndarray

nhập numpy dưới dạng np

lớp InfoArray(np.ndarray):

```
def __new__(subtype, hình dạng, dtype=float, buffer=None, offset=0,
          sải bước=Không có, đặt hàng=Không có, thông tin=Không có):
    # Tạo thẻ hiện ndarray của kiểu của chúng ta, với các đối số đầu vào # ndarray thông
    # thường. Điều này sẽ gọi hàm tạo #ndarray tiêu chuẩn, nhưng trả về một đối tượng thuộc
    # loại của chúng ta.
    # Nó cũng kích hoạt lệnh gọi tới InfoArray.__array_finalize__ obj =
    super(InfoArray, subtype).__new__(subtype, shape, dtype, buffer, offset, sải
                                      bước,
                                      đặt hàng)
    # đặt thuộc tính 'info' mới thành giá trị được truyền obj.info = info #
    Cuối cùng, chúng ta
    phải trả về đối tượng vừa tạo: return obj

def __array_finalize__(self, obj): # ``self`` là một
    # đối tượng mới được tạo ra từ # ndarray.__new__(InfoArray, ...),
    # do đó nó chỉ có # thuộc tính mà hàm tạo ndarray.__new__ cấp cho nó - # tức là những
    # thuộc tính đó của một ndarray tiêu chuẩn. ,
    # Chúng ta có thể nhận được lệnh gọi ndarray.__new__ theo 3 cách: # Từ một hàm tạo rõ ràng
    # - ví dụ InfoArray(): # obj là Không có (chúng ta đang ở giữa hàm tạo
    InfoArray.__new__ và
    ,      self.info sẽ là được đặt khi chúng ta quay lại InfoArray.__new__() nếu
    ,      obj là Không có: return # Từ truyền chế độ xem - ví dụ arr.view(InfoArray): # # # Từ
    # mẫu mới - ví dụ infoarr[:3]
    # type(obj) là InfoArray # # Lưu
    # ý rằng ở đây, thay vì trong phương thức __new__, # chúng ta đặt giá trị
    # obj là arr
    # (loại(obj) có thể là InfoArray)
    # mặc định cho 'thông tin', bởi vì phương thức # này nhìn thấy tất
    # cả việc tạo các đối tượng mặc định - với
    # hàm tạo # InfoArray.__new__, nhưng cũng với # arr.view(InfoArray). self.info = getattr(obj,
    # 'thông tin', Không có)

# Chúng tôi không cần trả lại bất cứ thứ gì
```

Sử dụng đối tượng trông như thế này:

```
>>> obj = InfoArray(shape=(3,)) # hàm tạo rõ ràng >>> type(obj) <class 'InfoArray'>
>>> obj.info Không
đúng

>>> obj = InfoArray(shape=(3,), info='information') >>> obj.info
'thông tin'
>>> v = obj[1:] # new-from-template - tại đây - cắt >>> type(v) <class 'InfoArray'>
>>> v.info

'thông tin'
>>> arr = np.arange(10) >>> cast_arr
= arr.view(InfoArray) # xem truyền >>> type(cast_arr) <class 'InfoArray'> >>>
cast_arr.info là Không có

ĐÚNG VẬY
```

Lớp này không hữu ích lắm, vì nó có cùng hàm tạo như đối tượng ndarray đơn giản, bao gồm cả việc truyền vào bộ đệm và hình dạng, v.v. Có lẽ chúng ta muốn hàm tạo có thể lấy một ndarray đã được hình thành từ các lệnh gọi numpy thông thường tới np.array và trả về một đối tượng.

### 3.9.7 Ví dụ thực tế hơn một chút - thuộc tính được thêm vào mảng hiện có

Đây là một lớp lấy một ndarray tiêu chuẩn đã tồn tại, chuyển thành kiểu của chúng tôi và thêm một thuộc tính bổ sung.

```
nhập numpy dưới dạng np

lớp RealisticInfoArray(np.ndarray):

    def __new__(cls, input_array, info=None):
        # Mảng đầu vào là một thẻ hiện ndarray đã được hình thành # Đầu tiên chúng ta ép
        # kiểu thành kiểu lớp obj = np.asarray(input_array).view(cls)
        # thêm thuộc tính mới vào thẻ hiện đã tạo obj.info = info #
        # Cuối cùng, chúng ta phải trả về đối tượng vừa tạo: return obj

    def __array_finalize__(self, obj): # xem
        InfoArray.__array_finalize__ để biết nhận xét nếu obj là Không: return
        self.info = getattr(obj, 'info',
                           None)
```

vì thế:

```
>>> arr = np.arange(5) >>> obj =
RealisticInfoArray(arr, info='information') >>> type(obj) <class 'RealisticInfoArray'>
>>> obj.info
'information'

>>> v = obj[1:] >>>
loại(v)
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
<class 'RealisticInfoArray'> >>> v.info
'thông tin'
```

### 3.9.8 \_\_array\_ufunc\_\_ cho ufunc

Mới trong phiên bản 1.13.

Một lớp con có thể ghi đè những gì xảy ra khi thực thi các ufuncs gọn gàng trên nó bằng cách ghi đè ndarray mặc định. \_\_phương thức \_\_mảng\_ufunc\_\_. Phương thức này được thực thi thay vì ufunc và sẽ trả về kết quả của thao tác hoặc NotImplemented . nếu thao tác được yêu cầu không được thực hiện.

Chữ ký của \_\_array\_ufunc\_\_ là:

```
def __array_ufunc__(ufunc, phương thức, *đầu vào, **kwargs):
    - *ufunc* là đối tượng ufunc được gọi. - *method* là một chuỗi cho biết
    Ufunc được gọi như thế nào, ``__call__`` để cho biết nó được gọi trực tiếp hoặc một trong
    các :ref:`methods<ufuncs.methods>`: ``reduce`` của nó ``, ``tích lũy`` ``, ``giảm`` ``, ``bên ngoài`` ``,
    hoặc ``tại`` ``.
    - *inputs* là một bộ các đối số đầu vào của ``ufunc`` - *kwargs* chứa bất kỳ đối số từ khóa hoặc
    tùy chọn nào được truyền cho hàm. Điều này bao gồm bất kỳ đối số ``out`` nào, luôn được chứa trong một bộ
    dữ liệu.
```

Một cách triển khai thông thường sẽ chuyển đổi bất kỳ đầu vào hoặc đầu ra nào là phiên bản của lớp của chính mình, chuyển mọi thứ sang siêu lớp bằng cách sử dụng super() và cuối cùng trả về kết quả sau khi có thể chuyển đổi ngược. Sau đây là một ví dụ được lấy từ trường hợp thử nghiệm test\_ufunc\_override\_with\_super trong core/tests/test\_umath.py.

```
nhập numpy như np

lớp A (np.ndarray):
    def __array_ufunc__(self, ufunc, phương thức, *đầu vào, **kwargs):
        args = []
        in_no = []
        đối với i, input_ trong liệt kê(đầu vào):
            nếu isinstance(input_, A):
                in_no.append(i)
            args.append(input_.view(np.ndarray)) khác:
                args.append(input_)

        đầu ra = kwargs.pop('out', None) out_no = [] if đầu
        ra: out_args =
        [] cho j, đầu ra
            ở dạng liệt kê(đầu
            ra):
                nếu isinstance(output, A):
                    out_no.append(j)
                    out_args.append(output.view(np.ndarray)) khác:
                        out_args.append(output) kwargs['out']
        = tuple(out_args) else:
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```

đầu ra = (Không có,) * ufunc.nout

info = {} if
in_no:
    info['inputs'] = in_no if out_no:
info['outputs']
    = out_no

kết quả = super(A, self).__array_ufunc__(ufunc, phương thức, *args, **kwargs)

nếu kết quả là NotImplemented: trả về
    NotImplemented

nếu phương thức == 'tại':
    if isinstance(inputs[0], A): đầu vào[0].info
        = info
    trả lại

nếu ufunc.nout == 1:
    kết quả = (kết quả,)

results = Tuple((np.asarray(result).view(A) nếu đầu ra là Không có đầu
    ra nào khác)
    đổi với kết quả, xuất ra dưới dạng zip(kết quả, kết quả đầu ra))
nếu kết quả và isinstance(results[0], A): results[0].info = info

trả về kết quả [0] nếu len(kết quả) == 1 kết quả khác

```

Vì vậy, lớp này thực sự không làm điều gì thú vị: nó chỉ chuyển đổi bất kỳ phiên bản nào của nó thành ndarray thông thường (nếu không, chúng ta sẽ có đệ quy vô hạn!), và thêm một từ điển thông tin cho biết đầu vào và đầu ra nào được chuyển đổi. Do đó, ví dụ,

```

>>> a = np.arange(5.).view(A) >>> b = np.sin(a)
>>> b.info

{'inputs': [0]} >>> b =
np.sin(np.arange(5.), out=(a,)) >>> b.info

{'outputs': [0]} >>> a =
np.arange(5.).view(A) >>> b = np.ones(1).view(A)
>>> c = a + b

>>> c.thông tin
{'inputs': [0, 1]} >>> a += b

>>> a.thông tin
{'đầu vào': [0, 1], 'đầu ra': [0]}

```

Lưu ý rằng một cách tiếp cận khác là sử dụng `getattr(ufunc, Methods)(*inputs, **kwargs)` thay vì gọi `super`. Trong ví dụ này, kết quả sẽ giống hệt nhau, nhưng có sự khác biệt nếu toán hạng khác cũng định nghĩa `__array_ufunc__`. Ví dụ: giả sử rằng chúng ta đánh giá `np.add(a, b)`, trong đó `b` là một phiên bản của lớp `B` khác có phần ghi đè. Nếu bạn sử dụng `super` như trong ví dụ, `ndarray.__array_ufunc__` sẽ nhận thấy rằng `b` có phần ghi đè, điều đó có nghĩa là nó không thể tự đánh giá kết quả. Do đó, nó sẽ trả về `NotImplemented` và lớp `A` của chúng ta cũng vậy. Sau đó, quyền kiểm soát sẽ được chuyển cho `b`, `b` biết cách xử lý chúng ta và tạo ra kết quả, hoặc không biết cách và trả về `NotImplemented`, gây ra `TypeError`.

Thay vào đó, nếu chúng ta thay thế lệnh gọi siêu của mình bằng `getattr(ufunc, phương thức)`, chúng ta sẽ thực hiện `np.add(a, view(np.ndarray))`, b một cách hiệu quả. Một lần nữa, `B.__array_ufunc__` sẽ được gọi, nhưng bây giờ nó coi `ndarray` là đối số còn lại. Có khả năng nó sẽ biết cách xử lý việc này và trả về một phiên bản mới của lớp B cho chúng ta. Lớp cũ của chúng tôi không được thiết lập để xử lý vấn đề này, nhưng nó có thể là cách tiếp cận tốt nhất nếu, ví dụ: một người triển khai lại `MaskedArray` bằng cách sử dụng `__array_ufunc__`.

Lưu ý cuối cùng: nếu siêu tuyến đường phù hợp với một lớp nhất định thì lợi thế của việc sử dụng nó là nó giúp xây dựng hệ thống phân cấp lớp. Ví dụ: giả sử rằng lớp B khác của chúng tôi cũng sử dụng siêu trong cách triển khai `__array_ufunc__` của nó và chúng tôi đã tạo một lớp C phụ thuộc vào cả hai, tức là lớp `C(A, B)` (để đơn giản, không phải ghi đè `__array_ufunc__` khác). Sau đó, bất kỳ `ufunc` nào trên phiên bản C sẽ chuyển sang `A.__array_ufunc__`, lệnh gọi siêu trong A sẽ chuyển đến `B.__array_ufunc__` và lệnh gọi siêu trong B sẽ chuyển đến `ndarray`. `__array_ufunc__`, do đó cho phép A và B cộng tác.

### 3.9.9 `__array_wrap__` cho ufuncs và các hàm khác

Trước numpy 1.13, hành vi của ufuncs chỉ có thể được điều chỉnh bằng cách sử dụng `__array_wrap__` và `__array_prepare__`. Hai cái này cho phép một người thay đổi loại đầu ra của `ufunc`, nhưng, ngược lại với `__array_ufunc__`, không cho phép một người thực hiện bất kỳ thay đổi nào đối với đầu vào. Người ta hy vọng rằng cuối cùng những thứ này sẽ không còn được dùng nữa, nhưng `__array_wrap__` cũng được sử dụng bởi các hàm và phương thức khó hiểu khác, chẳng hạn như `bop`, vì vậy tại thời điểm hiện tại vẫn cần thiết để có đầy đủ chức năng.

Về mặt khái niệm, `__array_wrap__` “kết thúc hành động” theo nghĩa cho phép một lớp con đặt loại giá trị trả về cũng như cập nhật các thuộc tính và siêu dữ liệu. Hãy cho thấy cách thức hoạt động của nó bằng một ví dụ. Đầu tiên chúng ta quay lại lớp con ví dụ đơn giản hơn, nhưng với một tên khác và một số câu lệnh in:

```
nhập numpy dưới dạng np

lớp MySubClass(np.ndarray):

    def __new__(cls, input_array, info=None):
        obj = np.asarray(input_array).view(cls) obj.info = trả về
        thông tin obj

    def __array_finalize__(self, obj):
        print('In __array_finalize__:') print(' self
        is %s' % repr(self)) print(' obj is %s' % repr(obj))
        if obj is None: return self.info = getattr(obj,
        'thông tin', Không có)

    def __array_wrap__(self, out_arr, context=None): print('In __array_wrap__:')
        print(' self is %s' % repr(self)) print(
        arr is %s' % repr(out_arr)) # then chỉ cần gọi cha mẹ
        return super(MySubClass, self).__array_wrap__(self,
        out_arr, context)
```

Chúng tôi chạy `ufunc` trên một phiên bản của mảng mới:

```
>>> obj = MySubClass(np.arange(5), info='spam')
Trong __array_finalize__:
    self là MySubClass([0, 1, 2, 3, 4]) obj là array([0,
    1, 2, 3, 4]) >>> arr2 = np.arange(5)+1 >>>
ret = np.add(arr2, obj)
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```

Trong __array_wrap__:
    self là MySubClass([0, 1, 2, 3, 4]) arr là array([1,
        3, 5, 7, 9])
Trong __array_finalize__:
    self là MySubClass([1, 3, 5, 7, 9]) obj là
    MySubClass([0, 1, 2, 3, 4]) >>> ret

MySubClass([1, 3, 5, 7, 9]) >>> ret.info

'thư rác'

```

Lưu ý rằng ufunc (np.add) đã gọi phương thức `__array_wrap__` với các đối số `self` là `obj` và `out_arr` là kết quả (`ndarray`) của phép cộng. Đổi lại, `__array_wrap__` (`ndarray. __array_wrap__`) mặc định đã chuyển kết quả sang lớp `MySubClass` và được gọi là `__array_finalize__` - do đó sẽ sao chép thuộc tính thông tin. Tất cả điều này đã xảy ra ở cấp độ C.

Tuy nhiên, chúng tôi có thể làm bất cứ điều gì chúng tôi muốn:

```

lớp SillySubClass(np.ndarray):

    def __array_wrap__(self, arr, context=None): trả về 'Tôi đã mất dữ
        liệu của bạn'

```

```

>>> arr1 = np.arange(5) >>> obj =
arr1.view(SillySubClass) >>> arr2 = np.arange(5)
>>> ret = np.multiply(obj, arr2)
>>> ret

'Tôi bị mất dữ liệu của bạn'

```

Vì vậy, bằng cách xác định một phương thức `__array_wrap__` cụ thể cho lớp con của mình, chúng ta có thể điều chỉnh đầu ra từ ufuncs. Phương thức `__array_wrap__` yêu cầu `self`, sau đó là một đối số - là kết quả của ufunc - và bối cảnh tham số tùy chọn. Tham số này được ufuncs trả về dưới dạng bộ dữ liệu 3 phần tử: (tên của ufunc, đối số của ufunc, miền của ufunc), nhưng không được thiết lập bởi các hàm numpy khác. Tuy nhiên, như đã thấy ở trên, bạn có thể làm khác, `__array_wrap__` sẽ trả về một thể hiện của lớp chứa nó. Xem lớp `mảng bị che` để biết cách triển khai.

Ngoài `__array_wrap__`, được gọi trên đường ra khỏi ufunc, còn có một phương thức `__array_prepare__` được gọi trên đường vào ufunc, sau khi các mảng đầu ra được tạo nhưng trước khi thực hiện bất kỳ tính toán nào. Việc triển khai mặc định không làm gì khác ngoài việc truyền qua mảng. `__array_prepare__` không nên cố gắng truy cập dữ liệu mảng hoặc thay đổi kích thước mảng, nó nhằm mục đích đặt loại mảng đầu ra, cập nhật thuộc tính và siêu dữ liệu cũng như thực hiện bất kỳ kiểm tra nào dựa trên đầu vào có thể mong muốn trước khi bắt đầu tính toán. Giống như `__array_wrap__`, `__array_prepare__` phải trả về một `ndarray` hoặc lớp con của nó hoặc gây ra lỗi.

### 3.9.10 Các vấn đề bổ sung - các phương thức `__del__` tùy chỉnh và `ndarray.base`

Một trong những vấn đề mà ndarray giải quyết là theo dõi quyền sở hữu bộ nhớ của ndarray và các khung nhín của chúng. Hãy xem xét trường hợp chúng ta đã tạo một ndarray, arr và lấy một lát cắt có `v = arr[1:]`. Hai đối tượng đang nhìn vào cùng một bộ nhớ. NumPy theo dõi dữ liệu đến từ đâu cho một mảng hoặc dạng xem cụ thể, với thuộc tính cơ sở:

```
>>> # Một ndarray bình thường, có dữ liệu riêng >>> arr
= np.zeros((4,))
>>> # Trong trường hợp này, base là None
>>> arr.base là None
ĐÚNG VẬY
>>> # Chúng tôi có một cái nhìn
>>> v1 = arr[1:] >>> #
base bây giờ trở đến mảng mà nó bắt nguồn từ >>> v1.base là arr
ĐÚNG VẬY
>>> # Ngắm nhìn một góc nhìn
>>> v2 = v1[1:] >>> #
diễn cơ sở cho khung nhín mà nó bắt nguồn từ đó
>>> v2.base là v1
ĐÚNG VẬY
```

Nói chung, nếu mảng sở hữu bộ nhớ riêng, như đối với arr trong trường hợp này, thì `arr.base` sẽ là Không - có một số trường hợp ngoại lệ đối với điều này - hãy xem cuốn sách gọn gàng để biết thêm chi tiết.

Thuộc tính `base` rất hữu ích trong việc có thể cho biết liệu chúng ta có chế độ xem hay mảng ban đầu. Điều này lại có thể hữu ích nếu chúng ta cần biết có nên thực hiện một số thao tác dọn dẹp cụ thể khi mảng phân lớp bị xóa hay không. Ví dụ: chúng tôi có thể chỉ muốn thực hiện việc dọn dẹp nếu mảng ban đầu bị xóa chứ không phải các khung nhín. Để biết ví dụ về cách hoạt động của tính năng này, hãy xem lớp `memmap` trong `numpy.core`.

### 3.9.11 Phân lớp và khả năng tương thích xuôi dòng

Khi phân lớp ndarray hoặc tạo các kiểu vit bắt chước giao diện ndarray, bạn có trách nhiệm quyết định mức độ liên kết giữa các API của bạn với các API có nhiều khối. Để thuận tiện, nhiều hàm numpy có phương thức ndarray tương ứng (ví dụ: tổng, trung bình, lấy, định hình lại) hoạt động bằng cách kiểm tra xem đối số đầu tiên của hàm có phương thức cùng tên hay không. Nếu nó tồn tại, phương thức này sẽ được gọi thay vì ép buộc các đối số vào một mảng có nhiều mảng.

Ví dụ: nếu bạn muốn lớp con hoặc kiểu vit của mình tương thích với hàm `tổng` của numpy, chữ ký phương thức cho phương thức `tổng` của đối tượng này phải như sau:

```
def sum(self, axis=None, dtype=None, out=None, keepdims=False):
    '
```

Đây chính xác là chữ ký phương thức tương tự cho `np.sum`, vì vậy bây giờ nếu người dùng gọi `np.sum` trên đối tượng này, numpy sẽ gọi phương thức tính `tổng` của chính đối tượng đó và chuyển vào các đối số được liệt kê ở trên trong chữ ký và sẽ không có lỗi nào phát sinh ... vì các chữ ký hoàn toàn tương thích với nhau.

Tuy nhiên, nếu bạn quyết định di chèch khỏi chữ ký này và làm điều gì đó như thế này:

```
tổng def (self, axis=None, dtype=None):
    '
```

Đối tượng này không còn tương thích với `np.sum` vì nếu bạn gọi `np.sum`, nó sẽ chuyển các đối số không mong muốn ra ngoài và giữ lại các đối số không mong muốn, khiến `TypeError` xuất hiện.

Nếu bạn muốn duy trì khả năng tương thích với numpy và các phiên bản tiếp theo của nó (có thể thêm các đối số từ khóa mới) nhưng không muốn hiển thị tất cả các đối số của numpy, thì chữ ký hàm của bạn phải chấp nhận `**kwargs`. Ví dụ:

```
tổng def (self, axis=None, dtype=None, **unused_kwargs):  
    '
```

Đối tượng này hiện đã tương thích lại với `np.sum` vì mọi đối số không liên quan (tức là các từ khóa không phải là `axis` hoặc `dtype`) sẽ bị ẩn trong tham số `**unused_kwargs`.



## ĐIỀU KHOẢN KHÁC

### 4.1 Giá trị đặc biệt của dấu phẩy động IEEE 754

Các giá trị đặc biệt được xác định trong numpy: nan, inf,

NaN có thể được sử dụng làm mặt nạ cho người nghèo (nếu bạn không quan tâm giá trị ban đầu là gì)

Lưu ý: không thể sử dụng đằng thức để kiểm tra NaN. Ví dụ:

```
>>> myarr = np.array([1., 0., np.nan, 3.]) >>>
np.nonzero(myarr == np.nan) (mảng[],
dtype=int64,) >>> np.nan ==
np.nan # luôn sai! Thay vào đó, hãy sử dụng các hàm numpy đặc biệt.
SAI
>>> myarr[myarr == np.nan] = 0. # không hoạt động
>>> myarr
array([ 1.,  0., 0., 3.]) >>> myarr[np.isnan(myarr)]
= 0. # sử dụng cái này thay vì find
>>> mảng
myarr ([ 1.,  0., 0., 3.])
```

Các hàm giá trị đặc biệt liên quan khác:

isinf():	Đúng nếu giá trị là inf
isfinite():	True nếu không nan hoặc inf nan_to_num():
Ánh xạ nan thành 0, inf thành float tối đa, -inf thành float tối thiểu	

Các hàm sau đây tương ứng với các hàm thông thường ngoại trừ việc nans bị loại khỏi kết quả:

```
nansum()
nanmax()
nanmin()
nanargmax()
nanargmin()

>>> x = np.arange(10.) >>>
x[3] = np.nan >>>
x.sum()
nan
>>> np.nansum(x) 42.0
```

## 4.2 Cách xử lý các ngoại lệ bằng số

Giá trị mặc định là 'cảnh báo' đối với trường hợp không hợp lệ, chia và tròn và 'bỏ qua' đối với trường hợp tràn. Nhưng điều này có thể được thay đổi và nó có thể được đặt riêng cho các loại ngoại lệ khác nhau. Các hành vi khác nhau là:

- 'bỏ qua' : Không thực hiện hành động nào khi ngoại lệ xảy ra.
- 'warn' : In RuntimeWarning ( thông qua cảnh báo Python mô-đun).
- 'raise' : Đưa ra lỗiFloatPointError.
- 'call' : Gọi một hàm được chỉ định bằng hàm seterrcall.
- 'print' : In cảnh báo trực tiếp ra thiết bị xuất chuẩn.
- 'log' : Ghi lại lỗi trong đối tượng Log được chỉ định bởi seterrcall.

Những hành vi này có thể được đặt cho tất cả các loại lỗi hoặc các lỗi cụ thể:

- all : áp dụng cho tất cả các ngoại lệ về số
- không hợp lệ : khi NaN được tạo
- chia : chia cho 0 (cho cả số nguyên!)
- tròn: tròn dấu phẩy động
- tròn dưới: dòng dưới dấu phẩy động

Lưu ý rằng phép chia số nguyên cho 0 được xử lý bởi cùng một máy móc. Những hành vi này được thiết lập trên cơ sở mỗi luồng.

## 4.3 Ví dụ

```
>>> oldsettings = np.seterr(all='warn') >>>
np.zeros(5,dtype=np.float32)/0. giá trị
không hợp lệ gấp phải trong phép chia
>>> j = np.seterr(under='ignore') >>>
np.array([1.e-100])**10 >>> j =
np.seterr(invalid='raise') >> >
np.sqrt(np.array([-1.]))
FloatPointError: giá trị không hợp lệ gấp phải trong sqrt >>> def
errorhandler(errstr, errflag): print("thấy lỗi ngu
'      ngốc!") >>> np.seterrcall(errorhandler)
<function err_handler at 0x...> >>> j =
np.seterr(all='call') >>> np.zeros(5,
dtype=np.int32)/0 FloatPointError: giá
tri không hợp lệ gấp phải trong phép chia đã
thấy lỗi ngu ngốc! >>> j = np.seterr(**oldsettings) # khôi phục # cài đặt xử lý
lỗi trước đó
'
```

## 4.4 Giao tiếp với C

Chỉ là một cuộc khảo sát về sự lựa chọn. Chi tiết nhỏ về cách mỗi hoạt động.

1) Kim loại tràn, bọc mã C của riêng bạn theo cách thủ công.

- Điểm cộng:

- Có hiệu quả \_

- Không phụ thuộc vào các công cụ khác

- Nhược điểm:

- Chi phí học tập nhiều:

- \* cần tìm hiểu cơ bản về Python C API

- \* cần tìm hiểu những điều cơ bản về API C gọn gàng

- \* cần học cách xử lý việc đếm tham chiếu và yêu thích nó.

Việc đếm tham chiếu thường khó thực hiện đúng.

- \* hiểu sai sẽ dẫn đến rò rỉ bộ nhớ và tệ hơn là lỗi phân đoạn

- API sẽ thay đổi cho Python 3.0!

2) Cython

- Điểm cộng:

- tránh học API C

- không xử lý việc đếm tham chiếu

- có thể viết mã bằng python giả và tạo mã C

- cũng có thể giao tiếp với mã C hiện có

- sẽ bảo vệ bạn khỏi những thay đổi đối với api Python C

- đã trở thành tiêu chuẩn thực tế trong cộng đồng Python khoa học

- hỗ trợ lập chỉ mục nhanh cho mảng

- Nhược điểm:

- Có thể viết code ở dạng không chuẩn, có thể tốn nén lỗi thời

- Không linh hoạt như gói thủ công

3) loại c

- Điểm cộng:

- một phần của thư viện chuẩn Python

- tốt cho việc giao tiếp với các thư viện có thể chia sẻ hiện có, đặc biệt là Windows DLL

- tránh các vấn đề về đếm API/tham chiếu

- hỗ trợ gọn gàng tốt: mảng có tất cả những thứ này trong thuộc tính `ctypes` của chúng:

<code>a.ctypes.data</code>	<code>a.ctypes.get_strides</code>
<code>a.ctypes.data_as</code>	<code>a.ctypes.shape</code>
<code>a.ctypes.get_as_parameter</code>	<code>a.ctypes.shape_as</code>

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

a.ctypes.get_data	a.ctypes.strides
a.ctypes.get_shape	a.ctypes.strides_as

- Nhược điểm:
  - không thể sử dụng để viết mã để chuyển thành phần mở rộng C, chỉ có công cụ bao bọc.

## 4) SWIG (trình tạo trình bao bọc tự động)

- Điểm cộng:
  - khoảng một thời gian dài
  - hỗ trợ nhiều ngôn ngữ kịch bản
  - Hỗ trợ C++
  - Tốt để gói các thư viện C lớn (nhiều chức năng) hiện có

- Nhược điểm:
  - tạo nhiều mã giữa Python và mã C
  - có thể gây ra các vấn đề về hiệu suất mà gần như không thể tối ưu hóa được
  - khó ghi các tập tin giao diện
  - không nhất thiết phải tránh các vấn đề về đếm tham chiếu hoặc cần biết API

## 5) scipy.weave

- Điểm cộng:
  - có thể biến nhiều biểu thức khó hiểu thành mã C
  - biên dịch động và tải mã C được tạo
  - có thể nhúng mã C thuận tiện vào mô-đun Python và có tính năng trích xuất dệt, tạo giao diện và biên dịch, v.v.

- Nhược điểm:
  - Tương lai rất không chắc chắn: đây là phần duy nhất của Scipy không được chuyển sang Python 3 và thực sự không được dùng nữa trong ứng hộ Cython.

## 6) Tâm lý

- Điểm cộng:
  - Biến python thuận thành mã máy hiệu quả thông qua tối ưu hóa giống như jit
  - rất nhanh khi nó tối ưu tốt

- Nhược điểm:
  - Chỉ có trên intel (windows?)
  - Không làm được gì nhiều cho Numpy à?

## 4.5 Giao diện với Fortran:

Sự lựa chọn rõ ràng để bọc mã Fortran là f2py.

Pyfort là một giải pháp thay thế cũ hơn nhưng không còn được hỗ trợ nữa. Fwrap là một dự án mới hơn có vẻ đầy hứa hẹn nhưng hiện không được phát triển nữa.

## 4.6 Giao tiếp với C++:

- 1) Cython
- 2) CXX
- 3) Boost.python
- 4) VÖI
- 5) SIP (được sử dụng chủ yếu trong PyQt)



## SÔ LƯỢNG CHO NGƯỜI SỬ DỤNG MATLAB

### 5.1 Giới thiệu

MATLAB® và NumPy/SciPy có nhiều điểm chung. Nhưng có nhiều sự khác biệt. NumPy và SciPy đã được tạo thực hiện tính toán số và khoa học theo cách tự nhiên nhất với Python, không phải là bản sao MATLAB®. Cái này Trang này nhằm mục đích thu thập kiến thức về sự khác biệt, chủ yếu nhằm mục đích giúp người thành thạo Người dùng MATLAB® trở thành người dùng NumPy và SciPy thành thạo.

### 5.2 Một số khác biệt chính

Trong MATLAB®, kiểu dữ liệu cơ bản là mảng đa chiều có dấu phẩy động có độ chính xác kép. Những con số. Hầu hết các biểu thức đều có các mảng như vậy và trả về mảng như vậy. Các thao tác trên phiên bản 2-D của các mảng này được thiết kế để hoạt động hiệu quả hơn hoặc ít hơn giống như các phép toán ma trận trong đại số tuyến tính.	Trong NumPy, kiểu cơ bản là mảng nhiều chiều. Các hoạt động trên các mảng này ở mọi chiều bao gồm cả 2D đều được thực hiện hoạt động theo từng phần tử. Người ta cần sử dụng các chức năng cụ thể đối với đại số tuyến tính (mặc dù đối với phép nhân ma trận, người ta có thể sử dụng toán tử @ trong python 3.5 trở lên).
MATLAB® sử dụng chỉ mục dựa trên 1 (một). Phần tử đầu tiên của chuỗi được tìm thấy bằng cách sử dụng a(1). <a href="#">nhìn thấy</a> lập chỉ mục ghi chú	Python sử dụng lập chỉ mục dựa trên 0 (không). Phần tử ban đầu của một chuỗi tự được tìm thấy bằng cách sử dụng a[0].
Ngôn ngữ kịch bản của MATLAB® được tạo ra cho làm đại số tuyến tính. Cú pháp cho các phép toán ma trận cơ bản rất hay và rõ ràng, nhưng API việc thêm GUI và tạo các ứng dụng chính thức ít nhiều là điều cần phải suy nghĩ lại.	NumPy dựa trên Python, được thiết kế ngay từ đầu để trở thành một ngôn ngữ lập trình đa năng xuất sắc. Trong khi cú pháp của Matlab cho một số thao tác mảng phức tạp hơn nhỏ gọn hơn NumPy, NumPy (do là một tiện ích bổ sung cho Python) có thể làm được nhiều việc mà Matlab không thể, chẳng hạn như ví dụ xử lý đúng cách với các ma trận.
Trong MATLAB®, các mảng có ngữ nghĩa theo từng giá trị, với sơ đồ sao chép khi ghi lười biếng để ngăn chặn việc thực sự tạo các bản sao cho đến khi chúng thực sự cần thiết. Các thao tác cắt lát sao chép các phần của mảng.	Trong mảng NumPy có ngữ nghĩa tham chiếu truyền qua. Các thao tác cắt lát được xem thành một mảng.

### 5.3 'mảng' hay 'ma trận'? Tôi nên sử dụng cái nào?

Về mặt lịch sử, NumPy đã cung cấp một loại ma trận đặc biệt, `np.matrix`, là một lớp con của `ndarray` dùng để thực hiện các phép toán nhị phân thành các phép toán đại số tuyến tính. Bạn có thể thấy nó được sử dụng trong một số mã hiện có thay vì `np.array`. Vì vậy, cái nào để sử dụng?

#### 5.3.1 Trả lời ngắn gọn

Sử dụng `mảng`.

- Chúng là loại `vector/ma trận/tensor` tiêu chuẩn của `numpy`. Nhiều hàm có nhiều mảng trả về mảng chứ không phải ma trận.
- Có sự phân biệt rõ ràng giữa các phép toán theo phần tử và các phép toán đại số tuyến tính.
- Bạn có thể có `vector` chuẩn hoặc `vector hàng/cột` nếu muốn.

Cho đến Python 3.5, nhược điểm duy nhất của việc sử dụng kiểu `mảng` là bạn phải sử dụng dấu chấm thay vì `*` để nhân (rút gọn) hai tensor (tích vô hướng, phép nhân `vector` ma trận, v.v.). Kể từ Python 3.5, bạn có thể sử dụng toán tử `@` nhân ma trận.

Với những điều trên, cuối cùng chúng tôi dự định sẽ không dùng ma trận nữa.

#### 5.3.2 Câu trả lời dài

`NumPy` chứa cả lớp `mảng` và lớp `ma trận`. Lớp `mảng` được dự định là một `mảng n chiều` có mục đích chung cho nhiều loại tính toán số, trong khi `ma trận` nhằm tạo điều kiện thuận lợi cho việc tính toán đại số tuyến tính cụ thể. Trong thực tế chỉ có một số khác biệt chính giữa hai loại này.

- Các toán tử `*` và `@`, các hàm `dot()`, và `multiple()`
  - Đối với `mảng`, `''` có nghĩa là phép nhân theo phần tử, trong khi `'@'` có nghĩa là phép nhân `ma trận`; chúng có các hàm liên quan `Multi()` và `dot()`. (Trước python 3.5, `@` không tồn tại và người ta phải sử dụng `dot()` để nhân `ma trận`).
  - Đối với hàm `''` có nghĩa là phép nhân `ma trận`, và để nhân theo phần tử, người ta phải sử dụng `ma trận`, hàm `multiple()`.
- Xử lý `vector` (`mảng` một chiều)
  - Đối với `mảng`, các hình dạng `vector 1xN`, `Nx1`, `N` đều khác nhau. Các phép toán như `A[:,1]` trả về `mảng` một chiều có hình `N`, không phải `mảng` hai chiều có hình `Nx1`. Chuyển đổi trên `mảng` một chiều không làm gì cả.
  - Đối với `ma trận`, `mảng` một chiều luôn được chuyển đổi lên thành `ma trận` `1xN` hoặc `Nx1` (`vector hàng` hoặc `cột`). `A[:,1]` trả về `ma trận` hai chiều có hình `Nx1`.
- Xử lý `mảng` nhiều chiều hơn (`ndim > 2`)
  - đối tượng `mảng` có thể có số chiều  $> 2$ ;
  - đối tượng `ma trận` luôn có chính xác hai chiều.
- Thuộc tính tiện lợi
  - `mảng` có thuộc tính `.T`, trả về sự chuyển vị của dữ liệu.
  - `ma trận` cũng có các thuộc tính `.H`, `.I` và `.A`, trả về hoán vị liên hợp, nghịch đảo và `asarray()` của `ma trận` tương ứng.
- Công cụ xây dựng tiện lợi

- Hàm tạo mảng lấy các chuỗi Python (lồng nhau) làm bộ khởi tạo. Như trong, `mảng([[1,2,3], [4,5,6]])`.
- Ngoài ra, hàm tạo ma trận còn có một trình khởi tạo chuỗi thuận tiện. Như trong `ma trận("[1 2 3; 4 5 6]")`.

Có những ưu và nhược điểm khi sử dụng cả hai:

' mảng

- :) Phép nhân phần tử rất dễ: `A*B`.
- :( Bạn phải nhớ rằng phép nhân ma trận có toán tử riêng là `@`.
- :) Bạn có thể coi mảng một chiều là vectơ hàng hoặc cột, trong khi `v @ A` coi `v` là vectơ hàng. Điều này có thể giúp bạn tiết kiệm được việc phải gõ rất nhiều chuyển vị.
- :) Mảng là loại NumPy "mặc định" nên được thử nghiệm nhiều nhất và là loại có nhiều khả năng xảy ra nhất. Được trả về bởi mã của bên thứ 3 sử dụng NumPy.
- :) Khá dễ dàng xử lý dữ liệu ở bất kỳ kích thước nào.
- :) Gần gũi hơn về mặt ngữ nghĩa với đại số tensor, nếu bạn đã quen với điều đó.
- :) Tất cả các thao tác `(* , / , + , - v.v.)` đều theo phần tử.
- :( Ma trận thua thớt từ `scipy.sparse` không tương tác tốt với mảng.

• ma trận

- :\ Hành vi giống với ma trận MATLAB® hơn.
- <:( Tôi đa là hai chiều. Để chứa dữ liệu ba chiều, bạn cần có mảng hoặc có thể là Python danh sách các ma trận.
- <:( Tôi thiếu là hai chiều. Bạn không thể có vectơ. Chúng phải được ép kiểu dưới dạng một cột hoặc ma trận một hàng.
- <:( Vì mảng là mặc định trong NumPy nên một số hàm có thể trả về một mảng ngay cả khi bạn cung cấp cho chúng một ma trận làm đối số. Điều này không xảy ra với các hàm NumPy (nếu đó là lỗi), nhưng mã của bên thứ 3 dựa trên NumPy có thể không tôn trọng việc bảo quản kiểu như NumPy.
- :) `A*B` là phép nhân ma trận nên trông giống như bạn viết trong đại số tuyến tính (Đối với Python `>= 3.5 plain` mảng có cùng sự thuận tiện với toán tử `@`).
- <:( Phép nhân theo phần tử yêu cầu gọi một hàm, `multiple(A,B)`.
- <:( Việc sử dụng nạp chồng toán tử hơi phi logic: `*` không hoạt động theo phần tử nhưng / thì có.
- Tương tác với `scipy.sparse` sạch sẽ hơn một chút.

Do đó, mảng này được khuyến khích sử dụng nhiều hơn. Thật vậy, cuối cùng chúng tôi dự định sẽ ngừng sử dụng ma trận.

## 5.4 Bảng tương đương MATLAB-NumPy

Bảng dưới đây cung cấp các giá trị tương đương cho một số biểu thức MATLAB® phổ biến. Đây không phải là những thông tin tương đương chính xác mà nên được coi là gợi ý để giúp bạn đi đúng hướng. Để biết thêm chi tiết, hãy đọc tài liệu tích hợp về các hàm NumPy.

Trong bảng bên dưới, giả định rằng bạn đã thực thi các lệnh sau bằng Python:

```
từ nhập numpy * nhập
scipy.linalg
```

Bên dưới cũng giả sử rằng nếu Ghi chú nói về "ma trận" thì các đối số là các thực thể hai chiều.

#### 5.4.1 Tương đương mục đích chung

Tạo giúp khó khăn về		Ghi chú
MATLAB vui vẻ	thông tin(func) hoặc trợ giúp(func) hoặc vui vẻ? (trong Ipython)	nhận trợ giúp về hàm func
cái mà vui vẻ	xem ghi chú GIÚP ĐỠ	tìm ra nơi func được xác định
các loại vui vẻ	nguồn(func) hay func?? (TRONG ipython)	nguồn in cho func (nếu không phải là hàm gốc)
một && b	A và B	toán tử logic AND ngắn mạch (bản địa Python nhà điều hành); chỉ đối số vô hướng
một    b	a hoặc b	toán tử OR logic ngắn mạch (toán tử gốc Python); chỉ đối số vô hướng
1*i, 1*j, 1i, 1j	1j	số phức
eps	np.spacing(1)	Khoảng cách giữa số 1 và số đầu phẩy động gần nhất.
ode45	scipy.integrate. giải_ivp(f)	tích hợp ODE với Runge-Kutta 4.5
ode15s	scipy.integrate. giải_ivp(f, phương thức='BDF')	tích hợp ODE với phương pháp BDF

#### 5.4.2 Đại số tuyến tính tương đương

MATLAB	NumPy
ndims(a)	ndim(a) hoặc a.ndim
số(a)	size(a) hoặc a.size
size(a)	hình dạng(a) hoặc a.shape
size(a,n) [ 1	a.shape[n-1]
2 3; 4 5 6 ] [ ab; cd ]	array([[1.,2.,3.], [4.,5.,6.]]) block([[a,b], [c,d]]) a[-1] a[1,4] a[1] hoặc
a(end) a(2,5)	a[1,:]
a(2,:)	a[0 :5]
a(1:5,:)	hoặc a[:5] hoặc a[0:5,:] a[-5:] a[0:3][:,4:9]
a(end-4:end,:)	a[4:-3,:]
a(1:3,5:9)	a[ix_([1,3,4], [ 0,2])] a[ 2:21:2,:,:] a[ ::2,:,:]
a([2 ,4,5],[1,3])	a[ ::-1,:,:]
a(3:2:21,:)	a[::len(a),0]]
a(1:2:end,:)	a.transpose() hoặc a.T
a(end:-1:1,:)	a.conj().transpose() hoặc a.conj().T
a.* b	a*b

MATLAB	NumPy a/
a ./b	b
a.^3	một**3
(a>0,5)	(a>0.5)
find(a>0,5)	khác không(a>0,5)
a(:,find(v>0,5))	a[:, nonzero(v>0,5)[0]] a[:, vT>,5]
a(:,find(v>0,5))	a[a<0,5]=0 a *
a(a<0,5)=0 a .*	(a>0,5) a[ :]
(a>0,5) a (:)=3	= 3 y = x.copy()
	y =
	x[1,:].copy() y =
y=xy=x(2,:)	x.flatten() arange(1.,11.)
y=x(:)	hoặc r_[1.:11.] hoặc
1:10	r_[ 1:10:10j] arange(10.) hoặc r_[:10.] hoặc r_[:9:10j]
0:9	arange(1.,11.):, newaxis] zeros((3,4))
[1:10]'	zeros(( 3,4,5)) one((3,4)) eye(3) diag(a)
số không(3,4)	diag(a,0)
số không(3,4,5)	Random.rand(3,4)
số một(3,4)	hoặc
mắt(3)	
diag(a)	
diag(a,0)	
rand(3,4)	Random.random_sample((3, 4)) linspace( 1,3,4) mgrid[0:9.,0:6.] hoặc
linspace(1, 3,4)	Meshgrid(r_[0:9.],r_[0:6.]
[x,y]=meshgrid(0:8,0:5)	ogrid[0:9.,0:6.] hoặc ix_(r_[0:9.],r_[0:6.] [x,y]=meshgrid([1,2,4], [2,4,5]) Meshgrid([1,2,4], [2,4,5])
	ix_([1,2,4],[2,4,5]) gạch(a,
Repmat(a, m, n) [ab]	(m, n) nối((a,b),1)
[a; b]	hoặc hstack((a,b)) hoặc cột_stack( (a,b)) nối((a,b)) hoặc vstack((a,b)) hoặc r_[a,b]
tối	a.max() a.max(0) a.max(1) max(a , b) sqrt(v @ v) hoặc
đa(tối đa(a))	
max(a)	
max(a,[],2)	
max(a,b)	np.linalg.norm(v)
Norm(v)	logic_and(a,b) logic_or(a,b)
a & b	
một	
b bitand(a,b)	a & b
bitor(a,b)	một
inv(a)	b linalg.inv(a)
pinv(a)	linalg.pinv(a)
xếp	linalg.matrix_rank(a)
	linalg.solve(a,b) nếu a là hình vuông; linalg.lstsq(a,b) nếu không thì giải aT
hạng(a) a\bb/a	xT = bT
[U,S,V]=svd(a)chol(a)	U, S, Vh = linalg.svd(a), V = Vh.T
	linalg.cholesky(a).TD,V =
[V,D]=eig(a)	linalg.eig(a)
[V,D]=eig(a,b)	D,V = scipy.linalg.eig(a,b)
[V,D]=eig(a,k)	

MATLAB	NumPy
[Q,R,P]=qr(a,0)	Q,R = scipy.linalg.qr(a)
[L,U,P]=lu(a) liên	L,U = scipy.linalg.lu(a) hoặc LU,P=scipy.linalg.lu_factor(a) scipy.sparse.linalg.cg fft(a)
hợp fft(a)	ifft(a).sort(a) hoặc a.sort()
ifft(a)	
sắp xếp(a)	
[b,I] =	
sắp xếp(a,i) hồi quy(y,X)	I = argsort(a[:,i]), b=a[I,:].linalg.lstsq(X,y)
decimate(x, q) duy	scipy.signal.resample(x,
nhất (a) siết chặt(a)	len(x)/q) duy nhất(a)
	a.bóp()

## 5.5 Ghi chú

Ma trận con: Việc gán cho ma trận con có thể được thực hiện bằng danh sách các chỉ mục bằng lệnh `ix_`. Ví dụ: đối với mảng 2d `a`, người ta có thể làm: `ind=[1,3]; a[np.ix_(ind,ind)]+=100`.

TRỢ GIÚP: Không có lệnh nào tương đương trực tiếp với lệnh `which` trong MATLAB, nhưng các lệnh trợ giúp và nguồn sẽ thường liệt kê tên tệp nơi chứa hàm. Python cũng có mô-đun `kiểm tra` (`kiểm tra nhập`) cung cấp một `getfile` thường hoạt động.

LẬP CHỈ MỤC: MATLAB® sử dụng một chỉ mục dựa trên một, do đó phần tử ban đầu của chuỗi có chỉ mục 1. Python sử dụng lập chỉ mục dựa trên 0, do đó phần tử ban đầu của chuỗi có chỉ số 0. Sự nhầm lẫn và chiến tranh bùng phát sinh bởi vì mỗi có những ưu điểm và nhược điểm. Việc lập chỉ mục một cơ sở phù hợp với cách sử dụng ngôn ngữ thông thường của con người, trong đó Phần tử "đầu tiên" của chuỗi có chỉ mục 1. Lập chỉ mục dựa trên 0 giúp đơn giản hóa việc lập chỉ mục. Xem thêm bài viết của prof.dr. Edsger W. Dijkstra.

PHẠM VI: Trong MATLAB®, `0:5` có thể được sử dụng làm chỉ mục cả bằng chữ và 'lát' (bên trong dấu ngoặc đơn); Tuy nhiên, trong Python, các cấu trúc như `0:5` chỉ có thể được sử dụng làm chỉ mục lát cắt (bên trong dấu ngoặc vuông). Vì thế có phần kỳ quặc đối tượng `x_` được tạo để cho phép numpy có cơ chế xây dựng phạm vi ngắn gọn tương tự. Lưu ý rằng `x_` không phải được gọi giống như một hàm hoặc một hàm tạo, nhưng được lập chỉ mục bằng dấu ngoặc vuông, cho phép sử dụng lát cắt của Python cú pháp trong các đối số.

LOGICOPS: & hoặc | trong NumPy là bit AND/OR, trong khi ở Matlab & và | là logic AND/OR. Sự khác biệt phải rõ ràng đối với bất kỳ ai có kinh nghiệm lập trình quan trọng. Cả hai có thể hoạt động giống nhau, nhưng có có những khác biệt quan trọng. Nếu bạn đã sử dụng & hoặc | các toán tử, bạn nên sử dụng ufuncs NumPy `logic_and`/`logic_or`. Sự khác biệt đáng chú ý giữa Matlab's và NumPy's & và | toán tử là:

- Đầu vào {0,1} không logic: Đầu ra của NumPy là AND theo bit của đầu vào. Matlab coi mọi giá trị khác 0 là 1 và trả về logic AND. Ví dụ `(3 & 4)` trong NumPy là 0, trong khi ở Matlab cả 3 và 4 đều được coi là logic đúng và `(3 & 4)` trả về 1.
- Độ ưu tiên: Toán tử & của NumPy có độ ưu tiên cao hơn các toán tử logic như < và >; Matlab là đảo ngược.

Nếu bạn biết mình có các đối số boolean, bạn có thể sử dụng các toán tử bitwise của NumPy, nhưng hãy cẩn thận với dấu ngoặc đơn, như thế này: `z = (x > 1) & (x < 2)`. Sự vắng mặt của các dạng toán tử NumPy của `logic_and` và `logic_or` là một hậu quả đáng tiếc của thiết kế của Python.

RESHAPE và LINEAR INDEXING: Matlab luôn cho phép truy cập các mảng đa chiều bằng cách sử dụng vô hướng hoặc các chỉ số tuyến tính, NumPy thì không. Các chỉ số tuyến tính rất phổ biến trong các chương trình Matlab, ví dụ `find()` trên kết quả trả về ma trận chúng, trong khi `find` của NumPy hoạt động khác. Khi chuyển đổi mã Matlab, trước tiên có thể cần phải định hình lại ma trận thành một chuỗi tuyến tính, thực hiện một số thao tác lập chỉ mục và sau đó định hình lại. Khi định hình lại (thường) tạo ra lượt xem trên cùng một bộ lưu trữ, có thể thực hiện việc này khá hiệu quả. Lưu ý rằng thứ tự quét được sử dụng bởi `reshape`

trong NumPy mặc định theo thứ tự 'C', trong khi Matlab sử dụng thứ tự Fortran. Nếu bạn chỉ đơn giản là chuyển đổi sang một chuỗi tuyến tính và ngược lại thì điều này không thành vấn đề. Nhưng nếu bạn đang chuyển đổi các hình dạng lại từ mã Matlab dựa trên thứ tự quét, thì mã Matlab này: `z = reshape(x,3,4);` sẽ trở thành `z = x.reshape(3,4,order='F').copy()` trong NumPy.

## 5.6 Tùy chỉnh môi trường của bạn

Trong MATLAB®, công cụ chính có sẵn cho bạn để tùy chỉnh môi trường là sửa đổi đường dẫn tìm kiếm với vị trí của các hàm yêu thích của bạn. Bạn có thể đặt các tùy chỉnh đó vào tập lệnh khởi động mà MATLAB sẽ chạy khi khởi động.

NumPy, hay đúng hơn là Python, có cơ sở vật chất tương tự.

- Để sửa đổi đường dẫn tìm kiếm Python của bạn nhằm bao gồm vị trí của các mô-đun của riêng bạn, hãy xác định PYTHONPATH. các biến môi trường.
- Để thực thi một tệp tập lệnh cụ thể khi khởi động trình thông dịch Python tương tác, hãy xác định biến môi trường PYTHONSTARTUP để chứa tên tệp lệnh khởi động của bạn.

Không giống như MATLAB®, nơi mọi thứ trên đường dẫn của bạn có thể được gọi ngay lập tức, với Python trước tiên bạn cần thực hiện câu lệnh 'nhập' để làm cho các hàm trong một tệp cụ thể có thể truy cập được.

Ví dụ: bạn có thể tạo một tập lệnh khởi động trông như thế này (Lưu ý: đây chỉ là một ví dụ, không phải là tuyên bố về "các phương pháp hay nhất"):

```
# Cung cấp tất cả các hàm numpy thông qua tiền tố 'np' ngắn hơn import numpy as np #
Làm cho tất cả các hàm matlib
có thể truy cập được ở cấp cao nhất thông qua M.func() import numpy.matlib as M # Tạo một số hàm matlib có thể truy
cập trực tiếp ở cấp cao nhất thông qua,
ví dụ: rand(3,3) from numpy.matlib import rand,zeros,ones,empty,eye # Xác định hàm Hermitian

def hermitian(A, **kwargs): return
    np.transpose(A,**kwargs).conj()
# Tạo một số phím tắt cho transpose,hermitian: # np.transpose(A) --> T(A) #
hermitian(A) --> H(A)

T = np.transpose H = ẩn sī
```

## 5.7 Liên kết

Xem <http://mathesaurus.sf.net/> để tham khảo chéo MATLAB®/NumPy khác.

Bạn có thể tìm thấy danh sách đầy đủ các công cụ dành cho công việc khoa học với python trên trang phần mềm chuyên đề.

MATLAB® và Simulink® là thương hiệu đã đăng ký của The MathWorks.



---

## XÂY DỰNG TỪ NGUỒN

Tổng quan chung về việc xây dựng NumPy từ nguồn được đưa ra ở đây, với các hướng dẫn chi tiết cho các nền tảng cụ thể được cung cấp riêng.

### 6.1 Điều kiện tiên quyết

Xây dựng NumPy yêu cầu cài đặt phần mềm sau:

1) Python 3.5.x hoặc mới hơn

Xin lưu ý rằng các tiêu đề phát triển Python cũng cần được cài đặt, ví dụ: trên Debian/Ubuntu, người ta cần cài đặt cả python3 và python3-dev. Trên Windows và macOS, điều này thường không phải là vấn đề.

2) Trình biên dịch

Để xây dựng bất kỳ mô-đun mở rộng nào cho Python, bạn sẽ cần trình biên dịch C. Nhiều mô-đun NumPy khác nhau sử dụng thư viện FORTRAN 77, vì vậy bạn cũng sẽ cần cài đặt trình biên dịch FORTRAN 77.

Lưu ý rằng NumPy được phát triển chủ yếu bằng trình biên dịch GNU. Các trình biên dịch từ các nhà cung cấp khác như Intel, Absoft, Sun, NAG, Compaq, Vast, Portland, Lahey, HP, IBM, Microsoft chỉ được hỗ trợ dưới dạng phản hồi của cộng đồng và có thể không hoạt động tốt. Nên sử dụng trình biên dịch GCC 4.x (và phiên bản mới hơn).

3) Thư viện đại số tuyến tính

NumPy không yêu cầu cài đặt bất kỳ thư viện đại số tuyến tính bên ngoài nào. Tuy nhiên, nếu những thứ này có sẵn, tập lệnh thiết lập của NumPy có thể phát hiện chúng và sử dụng chúng để xây dựng. Có thể sử dụng một số thiết lập thư viện LAPACK khác nhau, bao gồm các thư viện LAPACK được tối ưu hóa như OpenBLAS hoặc MKL.

4) Cython

Để xây dựng NumPy, bạn sẽ cần phiên bản Cython mới nhất.

### 6.2 Cài đặt cơ bản

Để cài đặt NumPy, hãy chạy:

cài đặt pip .

Để thực hiện bản dựng tại chỗ có thể chạy từ thư mục nguồn, hãy chạy:

python setup.py build\_ext --inplace

Lưu ý: để biết hướng dẫn xây dựng nhằm thực hiện công việc phát triển trên chính NumPy, hãy xem môi trường phát triển.

## 6.3 Kiểm tra

Hãy chắc chắn để kiểm tra các bản dựng của bạn. Để đảm bảo mọi thứ vẫn ổn định, hãy xem liệu tất cả các bài kiểm tra có vượt qua hay không:

```
$ python runtests.py -v -m đầy đủ
```

Để biết thông tin chi tiết về thử nghiệm, hãy xem các bản dựng thử nghiệm.

### 6.3.1 Bản dựng song song

Có thể thực hiện xây dựng song song với:

```
python setup.py build -j 4 cài đặt --prefix $HOME/.local
```

Điều này sẽ biên dịch numpy trên 4 CPU và cài đặt nó vào tiền tố đã chỉ định. Để thực hiện xây dựng song song tại chỗ, hãy chạy:

```
python setup.py build_ext --inplace -j 4
```

Số lượng công việc xây dựng cũng có thể được chỉ định thông qua biến môi trường NPY\_NUM\_BUILD\_JOBS.

### 6.3.2 Chọn trình biên dịch fortran

Trình biên dịch được tự động phát hiện; việc xây dựng bằng một trình biên dịch cụ thể có thể được thực hiện bằng --fcompiler. Ví dụ để chọn gfortran:

```
xây dựng python setup.py --fcompiler=gnu95
```

Để biết thêm thông tin xem:

```
xây dựng python setup.py --help-fcompiler
```

### 6.3.3 Cách kiểm tra ABI của thư viện BLAS/LAPACK

Một cách tương đối đơn giản và đáng tin cậy để kiểm tra trình biên dịch được sử dụng để xây dựng thư viện là sử dụng ldd trên thư viện. Nếu libg2c.so là phần phụ thuộc, điều này có nghĩa là g77 đã được sử dụng (lưu ý: g77 không còn được hỗ trợ để xây dựng NumPy).

Nếu libgfortran.so là phần phụ thuộc thì gfortran đã được sử dụng. Nếu cả hai đều phụ thuộc, điều này có nghĩa là cả hai đều đã được sử dụng, đây hầu như luôn là một ý tưởng rất tồi.

## 6.4 Thư viện BLAS/LAPACK được tăng tốc

NumPy tìm kiếm các thư viện đại số tuyến tính được tối ưu hóa như BLAS và LAPACK. Có các yêu cầu cụ thể để tìm kiếm các thư viện này như được mô tả bên dưới.

### 6.4.1 BLAS

Thứ tự mặc định cho các thư viện là:

1. MKL
2. HÀNH PHÚC
3. OpenBLAS
4. ATLAS
5. Tăng tốc (MacOS)
6. BLAS (NetLIB)

Nếu bạn muốn xây dựng dựa trên OpenBLAS nhưng bạn cũng có sẵn BLIS, bạn có thể xác định thứ tự tìm kiếm thông qua biến môi trường NPY\_BLAS\_ORDER, đây là danh sách các tên trên được phân tách bằng dấu phẩy, được sử dụng để xác định nội dung cần tìm kiếm, chẳng hạn:

```
NPY_BLAS_ORDER=ATLAS,blis,openblas,MKL python setup.py build
```

sẽ thích sử dụng ATLAS, sau đó là BLIS, sau đó là OpenBLAS và MKL là phương sách cuối cùng. Nếu cả hai điều này đều không tồn tại thì quá trình xây dựng sẽ thất bại (tên được so sánh bằng chữ thường).

### 6.4.2 LAPACK

Thứ tự mặc định cho các thư viện là:

1. MKL
2. OpenBLAS
3. libFLAME
4. ATLAS
5. Tăng tốc (MacOS)
6. LAPACK (NetLIB)

Ví dụ: nếu bạn muốn xây dựng dựa trên OpenBLAS nhưng bạn cũng có sẵn MKL, bạn có thể xác định thứ tự tìm kiếm thông qua biến môi trường NPY\_LAPACK\_ORDER, đây là danh sách các tên trên được phân tách bằng dấu phẩy:

```
NPY_LAPACK_ORDER=ATLAS,openblas,MKL python setup.py build
```

sẽ thích sử dụng ATLAS, sau đó là OpenBLAS và MKL là phương sách cuối cùng. Nếu cả hai điều này đều không tồn tại thì quá trình xây dựng sẽ thất bại (tên được so sánh bằng chữ thường).

### 6.4.3 Vô hiệu hóa ATLAS và các thư viện tăng tốc khác

Việc sử dụng ATLAS và các thư viện tăng tốc khác trong NumPy có thể bị vô hiệu hóa thông qua:

```
NPY_BLAS_ORDER= NPY_LAPACK_ORDER= python setup.py build
```

hoặc:

```
BLAS=Không có LAPACK=Không có ATLAS=Không có python setup.py build
```

#### 6.4.4 BLAS và LAPACK 64 bit

Bạn có thể yêu cầu Numpy sử dụng thư viện BLAS/LAPACK 64 bit bằng cách đặt biến môi trường:

```
NPY_USE_BLAS_ILP64=1
```

khi xây dựng Numpy. Các thư viện BLAS/LAPACK 64 bit sau được hỗ trợ:

1. OpenBLAS ILP64 có hậu tố ký hiệu 64\_ (openblas64\_)
2. OpenBLAS ILP64 không có hậu tố ký hiệu (openblas\_ilp64)

Thứ tự ưu tiên của chúng được xác định bởi các biến môi trường NPY\_BLAS\_ILP64\_ORDER và NPY\_LAPACK\_ILP64\_ORDER. Giá trị mặc định là openblas64\_, openblas\_ilp64.

---

Lưu ý: Việc sử dụng BLAS/LAPACK 64 bit không có hậu tố ký hiệu trong chương trình cũng sử dụng BLAS/LAPACK 32 bit có thể gây ra sự cố trong một số điều kiện nhất định (ví dụ: với trình thông dịch Python được nhúng trên Linux).

OpenBLAS 64-bit với hậu tố ký hiệu 64\_ có được bằng cách biên dịch OpenBLAS với các cài đặt:

```
tạo INTERFACE64=1 SYMBOLSUFFIX=64_
```

Hậu tố biểu tượng tránh xung đột tên biểu tượng giữa thư viện BLAS/LAPACK 32 bit và 64 bit.

---

#### 6.5 Cung cấp thêm cờ biên dịch

Cờ trình biên dịch bổ sung có thể được cung cấp bằng cách đặt các biến môi trường OPT, FOPT (cho Fortran) và CC. Khi cung cấp các tùy chọn giúp cải thiện hiệu suất của mã, hãy đảm bảo rằng bạn cũng đặt -DNDEBUG để mã gỡ lỗi không được thực thi.

---

## SỬ DỤNG C-API NUMPY

### 7.1 Cách mở rộng NumPy

Cái gì tinh tại và lặp đi lặp lại thì nhảm chán. Cái gì năng động và ngẫu nhiên  
thì khó hiểu. Ở giữa những lời đối trả là nghệ thuật.  
—John A. Locke

Khoa học là một phương trình vi phân. Tôn giáo là một điều kiện biên giới.  
- Alan Turing

#### 7.1.1 Viết mô-đun mở rộng

Mặc dù đối tượng ndarray được thiết kế để cho phép tính toán nhanh chóng trong Python, nhưng nó cũng được thiết kế cho mục đích chung và đáp ứng nhiều nhu cầu tính toán khác nhau. Kết quả là, nếu tốc độ tuyệt đối là điều cần thiết thì không có sự thay thế nào cho một vòng lặp được biên soạn kỹ lưỡng dành riêng cho ứng dụng và phần cứng của bạn. Đây là một trong những lý do khiến numpy bao gồm f2py để có sẵn cơ chế dễ sử dụng để liên kết mã C/C++ (đơn giản) và mã Fortran (tùy ý) trực tiếp vào Python. Bạn được khuyến khích sử dụng và cải thiện cơ chế này. Mục đích của phần này không phải là ghi lại công cụ này mà là ghi lại các bước cơ bản hơn để viết mô-đun mở rộng mà công cụ này phụ thuộc vào.

Khi một mô-đun mở rộng được viết, biên dịch và cài đặt vào một nơi nào đó trong đường dẫn Python (sys.path), mã sau đó có thể được nhập vào Python như thể nó là một tệp python tiêu chuẩn. Nó sẽ chứa các đối tượng và phương thức đã được định nghĩa và biên dịch bằng mã C. Các bước cơ bản để thực hiện việc này trong Python đã được ghi chép đầy đủ và bạn có thể tìm thêm thông tin trong tài liệu dành cho Python có sẵn trực tuyến tại [www.python.org](http://www.python.org).

Ngoài Python C-API, còn có C-API đầy đủ và phong phú cho NumPy cho phép thực hiện các thao tác phức tạp ở cấp độ C. Tuy nhiên, đối với hầu hết các ứng dụng, thông thường chỉ có một số lệnh gọi API được sử dụng. Nếu tất cả những gì bạn cần làm là trích xuất một con trỏ vào bộ nhớ cùng với một số thông tin hình dạng để chuyển sang một quy trình tính toán khác thì bạn sẽ sử dụng các lệnh gọi rất khác nhau, sau đó nếu bạn đang cố gắng tạo một kiểu mới giống như mảng hoặc thêm một dữ liệu mới .type cho ndarrays. Chương này ghi lại các lệnh gọi API và macro được sử dụng phổ biến nhất.

### 7.1.2 Chương trình con bắt buộc

Có chính xác một hàm phải được xác định trong mã C của bạn để Python sử dụng nó làm mô-đun mở rộng.

Hàm phải được gọi là `init{name}` trong đó `{name}` là tên của mô-đun trong Python. Hàm này phải được khai báo sao cho mã bên ngoài có thể nhìn thấy được. Bên cạnh việc thêm các phương thức và hằng số mà bạn mong muốn, chương trình con này cũng phải chứa các lệnh gọi như `import_array()` và/hoặc `import_ufunc()` tùy thuộc vào C-API nào cần thiết. Việc quên đặt các lệnh này sẽ tự hiển thị dưới dạng lỗi phân đoạn xấu (sự cố) ngay khi bắt kỳ chương trình con C-API nào thực sự được gọi. Thực tế có thể có nhiều hàm `init{name}` trong một tệp, trong trường hợp đó nhiều mô-đun sẽ được xác định bởi tệp đó. Tuy nhiên, có một số thủ thuật để làm cho nó hoạt động chính xác và nó không được đề cập ở đây.

Phương thức `init{name}` tối thiểu trông như sau:

```
PyMODINIT_FUNC
init{name}(void) {

    (void)Py_InitModule({name}, mymethods);
    import_array();
}
```

`Mymethod` phải là một mảng (thường được khai báo tĩnh) cấu trúc `PyMethodDef` chứa tên phương thức, hàm C thực tế, một biến cho biết phương thức đó có sử dụng đối số từ khóa hay không và chuỗi tài liệu.

Những điều này được giải thích trong phần tiếp theo. Nếu bạn muốn thêm hằng số vào mô-đun, thì bạn lưu trữ giá trị trả về từ `Py_InitModule`, một đối tượng mô-đun. Cách tổng quát nhất để thêm các mục vào mô-đun là lấy từ điển mô-đun bằng cách sử dụng `PyModule_GetDict(module)`. Với từ điển mô-đun, bạn có thể thêm bất cứ thứ gì bạn thích vào mô-đun theo cách thủ công. Một cách dễ dàng hơn để thêm đối tượng vào mô-đun là sử dụng một trong ba lệnh gọi Python C-API bổ sung mà không yêu cầu trích xuất riêng từ từ điển mô-đun. Những điều này được ghi lại trong tài liệu Python, nhưng được lặp lại ở đây để thuận tiện:

```
int PyModule_AddObject( mô-đun PyObject*, tên char*, PyObject* giá)
int PyModule_AddIntConstant( mô-đun PyObject*, tên char*, giá trị dài)
int PyModule>AddStringConstant( mô-đun PyObject*, tên char*, giá trị char*)
Cả ba hàm này đều yêu cầu đối tượng mô-đun (giá trị trả về của Py_InitModule). Tên là một chuỗi gắn nhãn giá trị trong mô-đun. Tùy thuộc vào hàm nào được gọi, đối số giá trị có thể là một đối tượng chung (PyModule_AddObject lấy tham chiếu đến nó), một hằng số nguyên hoặc một hằng chuỗi.
```

### 7.1.3 Xác định hàm

Đối số thứ hai được truyền vào hàm `Py_InitModule` là một cấu trúc giúp dễ dàng xác định các hàm trong mô-đun. Trong ví dụ nêu trên, cấu trúc `mymethods` lê ra đã được xác định trước đó trong tệp (thường là ngay trước chương trình con `init{name}`) để:

```
các phương thức tĩnh PyMethodDef tĩnh [] = {
    { nokeywordfunc,nokeyword_cfunc,
        METH_VAARGS,
        Chuỗi tài liệu},
    { keywordsfunc, keywords_cfunc,
        METH_VAARGS|METH_KEYWORDS,
        Chuỗi tài liệu},
    {NULL, NULL, 0, NULL} /* Trọng điểm */
},
```

Mỗi mục trong mảng `mymethods` là một `PyMethodDef` cấu trúc chứa 1) tên Python, 2) hàm C triển khai hàm, 3) cờ cho biết liệu từ khóa có được chấp nhận cho hàm này hay không và 4) Chuỗi tài liệu cho hàm. Bất kỳ số lượng chức năng nào cũng có thể được xác định cho một mô-đun bằng cách thêm nhiều mục vào mô-đun này

bàn. Mục nhập cuối cùng phải là tất cả NULL như được hiển thị để hoạt động như một trọng điểm. Python tìm mục này để biết rằng tất cả các hàm của mô-đun đã được xác định.

Điều cuối cùng phải làm để hoàn thành mô-đun mở rộng là viết mã thực hiện các chức năng mong muốn. Có hai loại hàm: loại không chấp nhận đối số từ khóa và loại chấp nhận.

Các hàm không có đối số từ khóa

Các hàm không chấp nhận đối số từ khóa phải được viết là:

```
tĩnh PyObject*
nokeyword_cfunc (PyObject *dummy, PyObject *args) {

    /* chuyển đổi các đối số Python */ /* thực
hiện hàm */ /* trả về một
cái gì đó */
,
```

Đối số giả không được sử dụng trong ngữ cảnh này và có thể được bỏ qua một cách an toàn. Đối số args chứa tất cả các đối số được truyền vào hàm dưới dạng một bộ dữ liệu. Bạn có thể làm bất cứ điều gì bạn muốn vào thời điểm này, nhưng thông thường cách dễ nhất để quản lý các đối số đầu vào là gọi `PyArg_ParseTuple` (`args, format_string,address_to_C_variables. . .`) hoặc `PyArg_UnpackTuple` (`tuple, "tên", tối thiểu, tối đa, . . .`). Mô tả hay về cách sử dụng hàm đầu tiên có trong số tay tham khảo Python C-API trong phần 5.5 (Phân tích đối số và xây dựng giá trị). Bạn nên đặc biệt chú ý đến định dạng "0&" sử dụng các hàm chuyển đổi để chuyển giữa đối tượng Python và đối tượng C. Tất cả các hàm định dạng khác có thể (hầu hết) được coi là trường hợp đặc biệt của quy tắc chung này. Có một số hàm chuyển đổi được xác định trong NumPy C-API có thể được sử dụng. Đặc biệt, hàm `PyArray_DescrConverter` rất hữu ích để hỗ trợ đặc tả kiểu dữ liệu tùy ý. Hàm này biến đổi bất kỳ đối tượng Python kiểu dữ liệu hợp lệ nào thành đối tượng `PyArray_Descr`. Hãy nhớ chuyển địa chỉ của các biến C cần diễn vào.

Có rất nhiều ví dụ về cách sử dụng `PyArg_ParseTuple` trong suốt mã nguồn NumPy. Cách sử dụng tiêu chuẩn là như thế này:

```
PyObject *đầu vào;
PyArray_Descr *dtype; if (!
PyArg_ParseTuple(args, "00&", &input,
                 PyArray_DescrConverter, &dtype))
    trả về NULL;
```

Điều quan trọng cần lưu ý là bạn nhận được một tham chiếu mượn đến đối tượng khi sử dụng chuỗi định dạng "0".

Tuy nhiên, các chức năng chuyển đổi thường yêu cầu một số dạng xử lý bộ nhớ. Trong ví dụ này, nếu chuyển đổi thành công, `dtype` sẽ giữ một tham chiếu mới đến đối tượng `PyArray_Descr`, trong khi đầu vào sẽ giữ một tham chiếu mượn. Do đó, nếu chuyển đổi này được trộn lẫn với một chuyển đổi khác (giả sử là số nguyên) và chuyển đổi kiểu dữ liệu thành công nhưng chuyển đổi số nguyên không thành công thì bạn cần phải giải phóng số tham chiếu cho đối tượng kiểu dữ liệu trước khi quay lại. Một cách điển hình để thực hiện việc này là đặt `dtype` thành `NULL` trước khi gọi `PyArg_ParseTuple` và sau đó sử dụng `Py_XDECREF` trên `dtype` trước khi quay lại.

Sau khi các đối số đầu vào được xử lý, mã thực sự thực hiện công việc sẽ được viết (có thể gọi các hàm khác nếu cần). Bước cuối cùng của hàm C là trả về một thứ gì đó. Nếu gặp lỗi thì phải trả về `NULL` (đảm bảo rằng lỗi thực sự đã được đặt). Nếu không có gì được trả về thì tăng `Py_None` và trả lại nó. Nếu một đối tượng được trả về thì nó sẽ được trả về (đảm bảo rằng bạn sở hữu một tham chiếu đến nó trước tiên). Nếu cần trả về nhiều đối tượng thì bạn cần trả về một bộ dữ liệu. Giá trị `Py_Build` (`format_string, c_variables. . .`) giúp dễ dàng xây dựng các bộ đối tượng Python từ các biến C. Đặc biệt chú ý đến sự khác biệt giữa 'N' và '0' trong chuỗi định dạng, nếu không bạn có thể dễ dàng tạo ra rò rỉ bộ nhớ. Chuỗi định dạng '0' tăng số lượng tham chiếu của `PyObject` \* Biến C mà nó tương ứng, trong khi chuỗi định dạng 'N' đánh cắp tham chiếu đến `PyObject` tương ứng \* biến C. Bạn nên sử dụng 'N' nếu bạn đã tạo tham chiếu cho đối tượng.

và chỉ muốn đưa tham chiếu đó đến bộ dữ liệu. Bạn nên sử dụng '0' nếu bạn chỉ có một tham chiếu mượn đến một đối tượng và cần tạo một tham chiếu để cung cấp cho bộ dữ liệu.

#### Các hàm có đối số từ khóa

Các hàm này rất giống với các hàm không có đối số từ khóa. Sự khác biệt duy nhất là chữ ký hàm là:

```
static PyObject*
keywords_cfunc (PyObject *dummy, PyObject *args, PyObject *kwds) {
    '
    '
```

Đối số kwds chứa một từ điển Python có khóa là tên của các đối số từ khóa và giá trị của nó là các giá trị đối số từ khóa tương ứng. Từ điển này có thể được xử lý theo cách bạn thấy phù hợp. Tuy nhiên, cách dễ nhất để xử lý nó là thay thế `PyArg_ParseTuple` (`args, format_string, address. . .`) với lệnh gọi tới `PyArg_ParseTupleAndKeywords` (`args, kwds, format_string, char *kwlist[], địa chỉ. . .`). Tham số kwlist cho hàm này là một mảng chuỗi kết thúc bằng NULL cung cấp các đối số từ khóa dự kiến. Cần có một chuỗi cho mỗi mục trong `format_string`. Việc sử dụng hàm này sẽ gây ra `TypeError` nếu đối số từ khóa không hợp lệ được truyền vào.

Để được trợ giúp thêm về chức năng này, vui lòng xem phần 1.8 (Tham số từ khóa cho hàm mở rộng) của hướng dẫn Mở rộng và nhúng trong tài liệu Python.

#### Đếm tham chiếu

Khó khăn lớn nhất khi viết module mở rộng là tính toán tham chiếu. Đó là lý do quan trọng giải thích sự phổ biến của f2py, dft, Cython, ctypes, v.v. , , , . Nếu bạn xử lý sai số lượng tham chiếu, bạn có thể gặp sự cố từ rò rỉ bộ nhớ đến lỗi phân đoạn. Chiến lược duy nhất tôi biết để xử lý số lượng tham chiếu một cách chính xác là máu, mồ hôi và nước mắt. Đầu tiên, bạn buộc phải nhớ rằng mọi biến Python đều có số lượng tham chiếu. Sau đó, bạn hiểu chính xác chức năng của từng hàm đối với số lượng tham chiếu của đối tượng để bạn có thể sử dụng DECREF và INCREF đúng cách khi cần. Việc đếm tham chiếu thực sự có thể kiểm tra mức độ kiên nhẫn và siêng năng mà bạn có đối với kỹ năng lập trình của mình. Bất chấp mô tả nghiêm ngã, hầu hết các trường hợp tính tham chiếu đều khá đơn giản với khó khăn phổ biến nhất là không sử dụng DECREF trên các đối tượng trước khi thoát sớm khỏi quy trình do một số lỗi. Vị trí thứ hai là lỗi phổ biến khi không sở hữu tham chiếu trên một đối tượng được truyền cho một hàm hoặc macro sẽ lấy cấp tham chiếu (ví dụ: `PyTuple_SET_ITEM`, và hầu hết các hàm lấy đối tượng `PyArray_Descr`).

Thông thường, bạn nhận được một tham chiếu mới đến một biến khi nó được tạo hoặc là giá trị trả về của một số hàm (tuy nhiên, có một số trường hợp ngoại lệ nổi bật - chẳng hạn như lấy một mục ra khỏi bộ dữ liệu hoặc từ điển). Khi bạn sở hữu tài liệu tham khảo, bạn có trách nhiệm đảm bảo rằng `Py_DECREF` (var) được gọi khi biến không còn cần thiết nữa (và không có hàm nào khác "dánh cắp" tham chiếu của nó). Ngoài ra, nếu bạn truyền một đối tượng Python cho một hàm sẽ "dánh cắp" tham chiếu, thì bạn cần đảm bảo rằng bạn sở hữu nó (hoặc sử dụng `Py_INCREF` để có được tài liệu tham khảo của riêng bạn). Bạn cũng sẽ gặp phải khái niệm mượn tài liệu tham khảo. Hàm mượn một tham chiếu sẽ không làm thay đổi số lượng tham chiếu của đối tượng và không mong muốn "giữ" tham chiếu. Nó sẽ chỉ sử dụng đối tượng tạm thời. Khi bạn sử dụng `PyArg_ParseTuple` hoặc `PyArg_UnpackTuple` bạn nhận được một tham chiếu mượn đến các đối tượng trong bộ dữ liệu và không được thay đổi số lượng tham chiếu của chúng bên trong hàm của bạn. Bằng cách thực hành, bạn có thể học cách đếm tham chiếu đúng cách, nhưng việc này có thể khiến bạn nản lòng lúc đầu.

Một nguồn lỗi đếm tham chiếu phổ biến là `Py_BuildValue` . chức năng. Hãy chú ý cẩn thận đến sự khác biệt giữa ký tự định dạng 'N' và ký tự định dạng '0'. Nếu bạn tạo một đối tượng mới trong chương trình con của mình (chẳng hạn như mảng đầu ra) và bạn chuyển nó trở lại thành một bộ giá trị trả về thì rất có thể bạn nên sử dụng ký tự định dạng 'N' trong `Py_BuildValue`. Ký tự '0' sẽ tăng số lượng tham chiếu lên một. Điều này sẽ để lại cho người gọi hai số lượng tham chiếu cho một mảng hoàn toàn mới. Khi biến bị xóa và số lượng tham chiếu giảm đi một, vẫn sẽ có số lượng tham chiếu bổ sung đó và mảng sẽ không bao giờ bị hủy phân bổ. Bạn sẽ

bị rò rỉ bộ nhớ do đếm tham chiếu. Sử dụng ký tự 'N' sẽ tránh được tình huống này vì nó sẽ trả về cho người gọi một đối tượng (bên trong bộ dữ liệu) với một số tham chiếu duy nhất.

#### 7.1.4 Xử lý các đối tượng mảng

Hầu hết các mô-đun mở rộng cho NumPy sẽ cần truy cập vào bộ nhớ cho đối tượng ndarray (hoặc một trong các lớp con của nó).

Cách dễ nhất để thực hiện việc này không yêu cầu bạn phải biết nhiều về nội bộ của NumPy. Phương pháp này là để

- Đảm bảo bạn đang xử lý một mảng hoạt động tốt (được căn chỉnh, theo thứ tự byte máy và một phân đoạn) của đúng loại và số lượng kích thước.
  - Bằng cách chuyển đổi nó từ một số đối tượng Python bằng PyArray\_FromAny hoặc macro được xây dựng trên nó.
  - Bằng cách xây dựng một ndarray mới có hình dạng và kiểu bạn mong muốn bằng cách sử dụng PyArray\_NewFromDescr hoặc một macro hoặc hàm đơn giản hơn dựa trên nó.
- Lấy hình dạng của mảng và con trỏ tới dữ liệu thực tế của mảng đó.
- Truyền dữ liệu và thông tin hình dạng vào chương trình con hoặc phần mã khác thực sự thực hiện tin học.
- Nếu bạn đang viết thuật toán, thì tôi khuyên bạn nên sử dụng thông tin bước tiến có trong mảng để truy cập các phần tử của mảng (macro PyArray\_GetPtr giúp việc này trở nên dễ dàng). Sau đó, bạn có thể nới lỏng các yêu cầu của mình để không ép buộc mảng một phân đoạn và có thể dẫn đến việc sao chép dữ liệu.

Mỗi chủ đề phụ này được đề cập trong các phần phụ sau.

##### Chuyển đổi một đối tượng chuỗi tùy ý

Quy trình chính để lấy một mảng từ bất kỳ đối tượng Python nào có thể được chuyển đổi thành một mảng là PyArray\_FromAny. Hàm này rất linh hoạt với nhiều đối số đầu vào. Một số macro giúp sử dụng chức năng cơ bản dễ dàng hơn. PyArray\_FROM\_OTF được cho là hữu ích nhất trong số các macro này cho các mục đích sử dụng phổ biến nhất. Nó cho phép bạn chuyển đổi một đối tượng Python tùy ý thành một mảng có kiểu dữ liệu dựng sẵn cụ thể (ví dụ: float), đồng thời chỉ định một bộ yêu cầu cụ thể (ví dụ: liền kề, căn chỉnh và có thể ghi). Cú pháp là

##### PyArray\_FROM\_OTF

Trả về một ndarray từ bất kỳ đối tượng Python nào, obj, có thể được chuyển đổi thành một mảng. Số lượng kích thước trong mảng trả về được xác định bởi đối tượng. Kiểu dữ liệu mong muốn của mảng trả về được cung cấp trong typenum, đây phải là một trong các kiểu được liệt kê. Các yêu cầu đối với mảng được trả về có thể là bất kỳ sự kết hợp nào của các cờ mảng tiêu chuẩn. Mỗi lập luận này sẽ được giải thích chi tiết hơn dưới đây. Bạn nhận được một tham chiếu mới tới mảng khi thành công. Nếu thất bại, NULL được trả về và một ngoại lệ được đặt.

vật thể

Đối tượng có thể là bất kỳ đối tượng Python nào có thể chuyển đổi thành ndarray. Nếu đối tượng đã là (một lớp con của) ndarray thỏa mãn yêu cầu thì một tham chiếu mới sẽ được trả về. Nếu không, một mảng mới sẽ được xây dựng. Nội dung của obj được sao chép sang mảng mới trừ khi giao diện mảng được sử dụng để dữ liệu không cần phải sao chép. Các đối tượng có thể được chuyển đổi thành một mảng bao gồm: 1) bất kỳ đối tượng chuỗi lồng nhau nào, 2) bất kỳ đối tượng nào hiển thị giao diện mảng, 3) bất kỳ đối tượng nào có phương thức \_\_array\_\_ (sẽ trả về một ndarray) và 4) bất kỳ đối tượng vô hướng nào (trở thành mảng không chiều). Các lớp con của ndarray phù hợp với yêu cầu sẽ được chuyển qua. Nếu bạn muốn đảm bảo ndarray lớp cơ sở, thì hãy sử dụng NPY\_ARRAY\_ENSUREARRAY trong cờ yêu cầu. Một bản sao chỉ được thực hiện nếu cần thiết. Nếu bạn muốn đảm bảo một bản sao, hãy chuyển NPY\_ARRAY\_ENSURECOPY vào cờ yêu cầu.

## kiểu chữ

Một trong các kiểu liệt kê hoặc NPY\_NOTYPE nếu kiểu dữ liệu phải được xác định từ chính đối tượng. Tên dựa trên C có thể được sử dụng:

```
NPY_BOOL, NPY_BYTE, NPY_UBYTE, NPY_SHORT, NPY USHORT, NPY_INT, NPY_UINT, NPY_LONG,
NPY ULONG, NPY_LONGLONG, NPY_ULONGLONG, NPY_DOUBLE, NPY_LONGLONG, NPY_CFLOAT,
NPY_CDOUBLE, NPY_CLONGDOUBLE, NPY_OBJECT.
```

Ngoài ra, tên độ rộng bit có thể được sử dụng như được hỗ trợ trên nền tảng. Ví dụ:

```
NPY_INT8, NPY_INT16, NPY_INT32, NPY_INT64, NPY_UINT8, NPY_UINT16, NPY_UINT32,
NPY_UINT64, NPY_FLOAT32, NPY_FLOAT64, NPY_COMPLEX64, NPY_COMPLEX128.
```

Đối tượng sẽ chỉ được chuyển đổi sang loại mong muốn nếu nó có thể được thực hiện mà không làm mất độ chính xác. Nếu không thì giá trị NULL sẽ được trả về và xuất hiện lỗi. Sử dụng NPY\_ARRAY\_FORCECAST trong cờ yêu cầu để ghi đè hành vi này.

## yêu cầu

Mô hình bộ nhớ cho ndarray thừa nhận các bước tùy ý trong mỗi chiều để tiến tới phần tử tiếp theo của mảng. Tuy nhiên, thông thường, bạn cần giao tiếp với mã yêu cầu bộ nhớ liền kề C hoặc liền kề Fortran. Ngoài ra, một ndarray có thể bị căn chỉnh sai (địa chỉ của một phần tử không phải là bội số nguyên của kích thước của phần tử đó), điều này có thể khiến chương trình của bạn bị lỗi (hoặc ít nhất là hoạt động chậm hơn) nếu bạn cố gắng hủy đăng ký một con trỏ vào dữ liệu mảng. Cả hai vấn đề này đều có thể được giải quyết bằng cách chuyển đổi đối tượng Python thành một mảng “hoạt động tốt” hơn cho mục đích sử dụng cụ thể của bạn.

Cờ yêu cầu cho phép xác định loại mảng nào được chấp nhận. Nếu đối tượng được truyền vào không đáp ứng yêu cầu này thì một bản sao sẽ được tạo để ba đối tượng được trả về sẽ đáp ứng yêu cầu. Những ndarray này có thể sử dụng một con trỏ rất chung tới bộ nhớ.

Cờ này cho phép đặc tả các thuộc tính mong muốn của đối tượng mảng được trả về. Tất cả các cờ được giải thích trong chương API chi tiết. Các cờ cần thiết nhất là NPY\_ARRAY\_IN\_ARRAY, NPY\_OUT\_ARRAY và NPY\_ARRAY\_INOUT\_ARRAY:

### NPY\_ARRAY\_IN\_ARRAY

Cờ này hữu ích cho các mảng phải theo thứ tự C liền kề và được căn chỉnh. Những loại mảng này thường là mảng đầu vào cho một số thuật toán.

### NPY\_ARRAY\_OUT\_ARRAY

Cờ này rất hữu ích để chỉ định một mảng theo thứ tự liền kề C, được căn chỉnh và cũng có thể được ghi vào. Một mảng như vậy thường được trả về dưới dạng đầu ra (mặc dù thông thường các mảng đầu ra như vậy được tạo từ đầu).

### NPY\_ARRAY\_INOUT\_ARRAY

Cờ này hữu ích để chỉ định một mảng sẽ được sử dụng cho cả đầu vào và đầu ra. PyArrayResolveWritebackIfCopy phải được gọi trước Py\_DECREF ở cuối quy trình giao diện để ghi lại dữ liệu tạm thời vào mảng ban đầu được truyền vào. Việc sử dụng cờ NPY\_ARRAY\_WRITEBACKIFCOPY hoặc NPY\_ARRAY\_UPDATEIFCOPY yêu cầu đối tượng đầu vào đã là một mảng (vì các đối tượng khác không thể được cập nhật tự động theo kiểu này). Nếu xảy ra lỗi, hãy sử dụng PyArray\_DiscardWritebackIfCopy (obj) trên một mảng có đặt các cờ này. Điều này sẽ đặt mảng cơ sở bên dưới có thể ghi mà không khiền nội dung bị sao chép trở lại mảng ban đầu.

Các cờ hữu ích khác có thể HOẶC theo yêu cầu bổ sung là:

**NPY\_ARRAY\_FORCECAST**

Truyền tới loại mong muốn, ngay cả khi không thể thực hiện được mà không làm mất thông tin.

**NPY\_ARRAY\_ENSURECOPY**

Hãy chắc chắn rằng kết quả là một bản sao của bản gốc.

**NPY\_ARRAY\_ENSUREARRAY**

Đảm bảo đối tượng thu được là một ndarray thực tế chứ không phải một lớp con.

---

Lưu ý: Việc một mảng có được hoán đổi byte hay không được xác định bởi kiểu dữ liệu của mảng. Mảng thứ tự byte gốc luôn được PyArray\_FROM\_OTF yêu cầu và do đó không cần gắn cờ NPY\_ARRAY\_NOTSWAPPED trong đối số yêu cầu. Cũng không có cách nào để có được một mảng hoán đổi byte từ quy trình này.

---

**Tạo một ndarray hoàn toàn mới**

Thông thường, các mảng mới phải được tạo từ bên trong mã mô-đun mở rộng. Có lẽ cần có một mảng đầu ra và bạn không muốn người gọi phải cung cấp nó. Có lẽ chỉ cần một mảng tạm thời để thực hiện phép tính trung gian. Bất kể nhu cầu là gì, đều có những cách đơn giản để có được đối tượng ndarray với bất kỳ kiểu dữ liệu nào cần thiết. Hàm chung nhất để thực hiện việc này là PyArray\_NewFromDescr. Tất cả các hàm tạo mảng đều trải qua mã được sử dụng lại nhiều này. Vì tính linh hoạt của nó, nó có thể hơi khó hiểu khi sử dụng. Kết quả là có những hình thức đơn giản hơn và dễ sử dụng hơn. Các biểu mẫu này là một phần của nhóm hàm PyArray\_SimpleNew, giúp đơn giản hóa giao diện bằng cách cung cấp các giá trị mặc định cho các trường hợp sử dụng phổ biến.

**Lấy bộ nhớ ndarray và truy cập các phần tử của ndarray**

Nếu obj là một ndarray (PyArrayObject \*), thì vùng dữ liệu của ndarray được trả tới bởi con trỏ void\* PyArray\_DATA (obj) hoặc con trỏ char\* PyArray\_BYTES (obj). Hãy nhớ rằng (nói chung) vùng dữ liệu này có thể không được căn chỉnh theo kiểu dữ liệu, nó có thể biểu thị dữ liệu bị hoán đổi byte và/hoặc nó có thể không ghi được. Nếu vùng dữ liệu được căn chỉnh và theo thứ tự byte gốc thì cách lấy một phần tử cụ thể của mảng chỉ được xác định bởi mảng các biến npy\_intp, PyArray\_STRIDES (obj). Cụ thể, mảng số nguyên c này cho biết số byte phải được thêm vào con trỏ phần tử hiện tại để đến phần tử tiếp theo trong mỗi chiều. Đối với các mảng nhỏ hơn 4 chiều, có các macro PyArray\_GETPTR{k} (obj, . . . ) trong đó {k} là số nguyên 1, 2, 3 hoặc 4 giúp việc sử dụng mảng trở nên dễ dàng hơn. Các lý lẽ . . . đại diện cho {k} chỉ số nguyên không âm vào mảng.

Ví dụ: giả sử E là mảng 3 chiều. Một con trỏ (void\*) tới phần tử E[i,j,k] được lấy dưới dạng PyArray\_GETPTR3 (E, i, j, k).

Như đã giải thích trước đây, mảng liền kề kiểu C và mảng liền kề kiểu Fortran có các kiểu sai bước cụ thể. Hai cờ mảng (NPY\_ARRAY\_C\_CONTIGUOUS và NPY\_ARRAY\_F\_CONTIGUOUS) cho biết liệu kiểu sai bước của một mảng cụ thể có khớp với kiểu liền kề kiểu C hay kiểu Fortran liền kề hay không. Mẫu sai bước có khớp với mẫu C hoặc Fortran tiêu chuẩn hay không, có thể được kiểm tra bằng cách sử dụng PyArray\_IS\_C\_CONTIGUOUS (obj) và PyArray\_ISFORTRAN (obj) tương ứng. Hầu hết các thư viện của bên thứ ba đều mong đợi các mảng liền kề. Tuy nhiên, thường thì không khó để hỗ trợ việc sai bước cho mục đích chung. Tôi khuyến khích bạn sử dụng thông tin sai bước trong mã của riêng bạn bắt cứ khi nào có thể và đặt trước các yêu cầu về một phân đoạn để gói mã của bên thứ ba. Việc sử dụng thông tin sai bước được cung cấp cùng với ndarray thay vì yêu cầu sai bước liền kề sẽ làm giảm việc sao chép mà lẽ ra phải thực hiện.

### 7.1.5 Ví dụ

Ví dụ sau đây cho thấy cách bạn có thể viết một trình bao bọc chấp nhận hai đối số đầu vào (sẽ được chuyển đổi thành một mảng) và một đối số đầu ra (phải là một mảng). Hàm trả về Không có và cập nhật mảng đầu ra. Lưu ý việc sử dụng ngữ nghĩa WRITEBACKIFCOPY được cập nhật cho NumPy v1.14 trở lên.

```
PyObject *tinh *
example_wrapper(PyObject *dummy, PyObject *args {

    PyObject *arg1=NULL, *arg2=NULL, *out=NULL;
    PyObject *arr1=NULL, *arr2=NULL, *oarr=NULL;

    if (!PyArg_ParseTuple(args, "OOO!", &arg1, &arg2, &PyArray_Type, &out)) trả
        về NULL;

    mảng1 = PyArray_FROM_OTF(arg1, NPY_DOUBLE, NPY_ARRAY_IN_ARRAY); if (arr1 == NULL) trả về NULL;
    mảng2 = PyArray_FROM_OTF(arg2, NPY_DOUBLE,
        NPY_ARRAY_IN_ARRAY); if (arr2 == NULL) bị lỗi; #if NPY_API_VERSION >= 0x0000000c

    oarr = PyArray_FROM_OTF(ra, NPY_DOUBLE, NPY_ARRAY_INOUT_ARRAY2); #khác

    oarr = PyArray_FROM_OTF(ra, NPY_DOUBLE, NPY_ARRAY_INOUT_ARRAY); #endif if (oarr == NULL) bị lỗi;

/* mã sử dụng các đối số */ /* Có thể bạn sẽ cần ít nhất
nd = PyArray_NDIM(<..>) dims = PyArray_DIMS(<..>
    -- mảng npy_intp có độ dài nd hiển      --số lượng kích thước
    thị độ dài trong mỗi dim.

    dptr = (double *)PyArray_DATA(<..>) -- con trỏ tới dữ liệu.

    Nếu xảy ra lỗi thì phải thắt bại. ,

    Py_DECREF(arr1);
    Py_DECREF(arr2); #if
NPY_API_VERSION >= 0x0000000c
    PyArray_ResolveWritebackIfCopy(oarr);
#endif

    Py_DECREF(oarr);
    Py_INCREF(Py_None); trả về
    Py_None;

    thắt bại:
    Py_XDECREF(arr1);
    Py_XDECREF(arr2); #if
NPY_API_VERSION >= 0x0000000c
    PyArray_DiscardWritebackIfCopy(oarr); #endif

    Py_DECREF(oarr); trả về
    NULL;
}
```

## 7.2 Sử dụng Python làm keo

Không có cuộc trò chuyện nào nhảm chán hơn cuộc trò chuyện mà mọi người đều đồng ý.

-Michel de Montaigne

Băng keo giống như lực. Nó có mặt sáng và mặt tối, và nó giữ vũ trụ lại với nhau.

-Carl Zwanzig

Nhiều người thích nói rằng Python là một ngôn ngữ kết dính tuyệt vời. Hy vọng rằng chương này sẽ thuyết phục bạn rằng điều này là đúng. Những người đầu tiên áp dụng Python cho khoa học thường là những người sử dụng nó để dán các mã ứng dụng lớn chạy trên siêu máy tính lại với nhau. Việc viết mã bằng Python không chỉ đẹp hơn nhiều so với tập lệnh shell hoặc Perl, ngoài ra, khả năng mở rộng Python dễ dàng khiêm việc tạo các lớp và kiểu mới phù hợp cụ thể với các vấn đề đang được giải quyết tương đối dễ dàng. Từ sự tương tác của những người đóng góp ban đầu này, Numeric nổi lên như một đối tượng dạng mảng có thể được sử dụng để truyền dữ liệu giữa các ứng dụng này.

Khi Numeric đã trưởng thành và phát triển thành NumPy, mọi người có thể viết nhiều mã hơn trực tiếp trong NumPy.

Thường thì mã này đủ nhanh để sử dụng trong sản xuất nhưng đôi khi vẫn cần truy cập vào mã đã biên dịch.

Hoặc để tận dụng chút hiệu quả cuối cùng của thuật toán hoặc để giúp truy cập các mã có sẵn rộng rãi được viết bằng C/C++ hoặc Fortran dễ dàng hơn.

Chương này sẽ xem xét nhiều công cụ có sẵn cho mục đích truy cập mã được viết bằng các ngôn ngữ biên dịch khác. Có rất nhiều tài nguyên có sẵn để học cách gọi các thư viện được biên dịch khác từ Python và mục đích của Chương này không phải để giúp bạn trở thành chuyên gia. Mục tiêu chính là giúp bạn biết về một số khả năng để bạn biết phải "Google" những gì để tìm hiểu thêm.

### 7.2.1 Gọi các thư viện đã biên dịch khác từ Python

Mặc dù Python là một ngôn ngữ tuyệt vời và rất thú vị khi viết mã, nhưng bản chất năng động của nó dẫn đến chi phí cao có thể khiến một số mã (tức là các tính toán thô bạo trong vòng lặp for) chậm hơn 10-100 lần so với mã tương đương được viết bằng ngôn ngữ biên dịch tĩnh . . Ngoài ra, nó có thể khiến mức sử dụng bộ nhớ lớn hơn mức cần thiết do các mảng tạm thời được tạo và hủy trong quá trình tính toán. Đối với nhiều loại nhu cầu tính toán, thường không thể tránh khỏi tình trạng chậm lại và tiêu thụ bộ nhớ nhiều hơn (ít nhất là đối với các phần quan trọng về thời gian hoặc bộ nhớ trong mã của bạn). Do đó, một trong những nhu cầu phổ biến nhất là gọi từ mã Python sang quy trình mã máy nhanh (ví dụ: được biên dịch bằng C/C++ hoặc Fortran). Thực tế là điều này tương đối dễ thực hiện là lý do chính tại sao Python là ngôn ngữ cấp cao tuyệt vời cho lập trình khoa học và kỹ thuật.

Đây là hai cách tiếp cận cơ bản để gọi mã được biên dịch: viết mô-đun mở rộng sau đó được nhập vào Python bằng lệnh nhập hoặc gọi trực tiếp chương trình con thư viện dùng chung từ Python bằng cách sử dụng ctypes . mô-đun.

Viết một mô-đun mở rộng là phương pháp phổ biến nhất.

**Cảnh báo:** Việc gọi mã C từ Python có thể dẫn đến lỗi Python nếu bạn không cẩn thận. Không có cách tiếp cận nào trong chương này là miễn dịch. Bạn phải biết điều gì đó về cách xử lý dữ liệu của cả NumPy và thư viện bên thứ ba đang được sử dụng.

### 7.2.2 Trình bao bọc được tạo bằng tay

Các mô-đun mở rộng đã được thảo luận trong [phần Viết mô-đun mở rộng](#). Cách cơ bản nhất để giao tiếp với mã đã biên dịch là viết một mô-đun mở rộng và xây dựng một phương thức mô-đun gọi mã đã biên dịch. Để cải thiện khả năng đọc, phương thức của bạn nên tận dụng lệnh gọi `PyArg_ParseTuple` để chuyển đổi giữa các đối tượng Python và kiểu dữ liệu C. Đối với các kiểu dữ liệu C tiêu chuẩn có thể đã có sẵn bộ chuyển đổi tích hợp. Đối với những người khác, bạn có thể cần phải viết trình chuyển đổi của riêng mình và sử dụng chuỗi định dạng "`O&`" cho phép bạn chỉ định một hàm sẽ được sử dụng để thực hiện chuyển đổi từ đối tượng Python sang bất kỳ cấu trúc C nào cần thiết.

Khi việc chuyển đổi sang cấu trúc C và kiểu dữ liệu C thích hợp đã được thực hiện, bước tiếp theo trong trình bao bọc là gọi hàm cơ bản. Điều này đơn giản nếu hàm cơ bản nằm trong C hoặc C++. Tuy nhiên, để gọi mã Fortran, bạn phải làm quen với cách gọi các chương trình con Fortran từ C/C++ bằng trình biên dịch và nền tảng của bạn. Điều này có thể thay đổi đôi chút nền tảng và trình biên dịch (đó là một lý do khác khiến f2py làm cho việc giao tiếp mã Fortran trở nên đơn giản hơn nhiều) nhưng nhìn chung liên quan đến việc xóa trộn tên gạch dưới và thực tế là tất cả các biến đều được truyền bằng tham chiếu (tức là tất cả các đối số đều là con trỏ).

Ưu điểm của trình bao bọc được tạo bằng tay là bạn có toàn quyền kiểm soát cách thư viện C được sử dụng và gọi, điều này có thể dẫn đến giao diện gọn gàng và chặt chẽ với chi phí tối thiểu. Điểm bất lợi là bạn phải viết, gỡ lỗi và duy trì mã C, mặc dù hầu hết mã này có thể được điều chỉnh bằng cách sử dụng kỹ thuật "cắt-dán-và-sửa đổi" từ các mô-đun mở rộng khác. Bởi vì quy trình gọi mã C bổ sung khá đơn giản nên các quy trình tạo mã đã được phát triển để làm cho quy trình này trở nên dễ dàng hơn. Một trong những kỹ thuật tạo mã này được phân phối với NumPy và cho phép tích hợp dễ dàng với mã Fortran và mã C (đơn giản). Gói này, f2py, sẽ được đề cập ngắn gọn trong phần tiếp theo.

### 7.2.3 f2py

F2py cho phép bạn tự động xây dựng một mô-đun mở rộng có giao diện với các quy trình trong mã Fortran 77/90/95. Nó có khả năng phân tích mã Fortran 77/90/95 và tự động tạo chữ ký Python cho các chương trình con mà nó gặp hoặc bạn có thể hướng dẫn cách chương trình con giao tiếp với Python bằng cách xây dựng tệp định nghĩa giao diện (hoặc sửa đổi tệp do f2py tạo ra).

#### Tạo nguồn cho mô-đun mở rộng cơ bản

Có lẽ cách dễ nhất để giới thiệu f2py là đưa ra một ví dụ đơn giản. Đây là một trong các chương trình con có trong tệp có tên `add.f`:

```
C
      CHƯƠNG TRÌNH CON ZADD(A,B,C,N)
C
      TỔNG HỢP NHÂN ĐÔI A(*)
      PHỨC HỢP NHÂN ĐÔI B(*)
      PHỨC HỢP NHÂN ĐÔI C(*)
      số nguyên N
      LÀM 20 J = 1, N
          C(J) = A(J)+B(J)
20 TIẾP TỤC
      KẾT THÚC
```

Quy trình này chỉ đơn giản là cộng các phần tử trong hai mảng liền kề và đặt kết quả vào mảng thứ ba. Bộ nhớ cho cả ba mảng phải được cung cấp bởi quy trình gọi. Một giao diện rất cơ bản cho quy trình này có thể được tạo tự động bởi f2py:

```
f2py -m thêm add.f
```

Bạn sẽ có thể chạy lệnh này nếu đường dẫn tìm kiếm của bạn được thiết lập đúng cách. Lệnh này sẽ tạo ra một mô-đun mở rộng có tên addmodule.c trong thư mục hiện tại. Mô-đun mở rộng này hiện có thể được biên dịch và sử dụng từ Python giống như bất kỳ mô-đun mở rộng nào khác.

#### Tạo mô-đun mở rộng được biên dịch

Bạn cũng có thể lấy f2py để biên dịch add.f và cũng biên dịch mô-đun mở rộng được tạo ra của nó, chỉ để lại tệp mở rộng thư viện dùng chung có thể được nhập từ Python:

```
f2py -c -m thêm add.f
```

Lệnh này để lại một tệp có tên add.{ext} trong thư mục hiện tại (trong đó {ext} là phần mở rộng thích hợp cho mô-đun mở rộng python trên nền tảng của bạn - vì vậy, pyd, v.v.). Mô-đun này sau đó có thể được nhập từ Python. Nó sẽ chứa một phương thức cho mỗi chương trình con trong add (zadd, cadd, Dadd, sadd). Chuỗi tài liệu của mỗi phương thức chứa thông tin về cách gọi phương thức mô-đun:

```
>>> nhập thêm >>> in
add.zadd.__doc__ zadd - Chữ ký hàm:

    cộng (a,b,c,n)
Các đối số bắt buộc:
    a : đầu vào mảng hạng 1('D') có giới hạn (*) b : đầu vào mảng
        hạng 1('D') có giới hạn (*) c : đầu vào mảng hạng 1('D') có giới
        hạn (*) n : nhập int
```

#### Cải thiện giao diện cơ bản

Giao diện mặc định là bản dịch rất đúng nghĩa của mã Fortran sang Python. Các đối số mảng Fortran bây giờ phải là mảng NumPy và đối số số nguyên phải là số nguyên. Giao diện sẽ cố gắng chuyển đổi tất cả các đối số thành loại (và hình dạng) được yêu cầu và đưa ra lỗi nếu không thành công. Tuy nhiên, vì nó không biết gì về ngữ nghĩa của các đối số (như C là đầu ra và n phải thực sự khớp với kích thước mảng), nên có thể lạm dụng hàm này theo những cách có thể khiến Python gặp sự cố. Ví dụ:

```
>>> add.zadd([1,2,3], [1,2], [3,4], 1000)
```

sẽ gây ra sự cố chương trình trên hầu hết các hệ thống. Trong phần bìa, các danh sách đang được chuyển đổi thành các mảng thích hợp nhưng sau đó, vòng lặp bổ sung cơ bản được yêu cầu quay vòng vượt ra ngoài ranh giới của bộ nhớ được phân bổ.

Để cải thiện giao diện, cần cung cấp các chỉ thị. Điều này được thực hiện bằng cách xây dựng một tệp định nghĩa giao diện. Thông thường, tốt nhất là bắt đầu từ tệp giao diện mà f2py có thể tạo ra (nơi nó có hành vi mặc định từ đó). Để f2py tạo tệp giao diện, hãy sử dụng tùy chọn -h:

```
f2py -h add.pyf -m thêm add.f
```

Lệnh này để lại tệp add.pyf trong thư mục hiện tại. Phần của tệp này tương ứng với zadd là:

```
chương trình con zadd(a,b,c,n) ! trong:thêm:add.f
    chiều phức tạp (*) :: a chiều phức tạp
    kép(*) :: b chiều phức tạp kép(*) :: c số
    nguyên :: n

kết thúc chương trình con zadd
```

Bằng cách đặt các chỉ thị ý định và kiểm tra mã, giao diện có thể được dọn dẹp khá nhiều cho đến khi phương thức mô-đun Python vừa dễ sử dụng hơn vừa mạnh mẽ hơn.

```
chương trình con zadd(a,b,c,n) ! trong: thêm:add.f
chiều phức tạp kép(n) :: một chiều phức tạp
kép(n) :: b ý định phức tạp kép(out), chiều(n) :: c
ý định số nguyên(ẩn), depend(a) :: n=len(a)

kết thúc chương trình con zadd
```

Chỉ thị ý định, ý định (ngoài) được sử dụng để cho f2py biết rằng c là một biến đầu ra và phải được giao diện tạo ra trước khi được chuyển đến mã cơ bản. Lệnh Intent(hide) yêu cầu f2py không cho phép người dùng chỉ định biến n mà thay vào đó lấy biến từ kích thước của a. Chỉ thị dependency(a) là cần thiết để cho f2py biết rằng giá trị của n phụ thuộc vào đầu vào a (để nó không cố tạo biến n cho đến khi biến a được tạo).

Sau khi sửa đổi add.pyf, tệp mô-đun python mới có thể được tạo bằng cách biên dịch cả add.f95 và add.pyf:

```
f2py -c add.pyf add.f95
```

Giao diện mới có docstring:

```
>>> nhập thêm >>> in
add.zadd.__doc__ zadd - Chữ ký hàm:

    c = zadd(a,b)
Các đối số bắt buộc:
    a : mảng đầu vào hạng 1('D') có giới hạn (n) b : mảng đầu vào hạng
        1('D') có giới hạn (n)
Trả về đối tượng: c :
    mảng hạng 1('D') có giới hạn (n)
```

Bây giờ, hàm này có thể được gọi theo cách mạnh mẽ hơn nhiều:

```
>>> add.zadd([1,2,3],[4,5,6]) array([ 5.+0.j,
    7.+0.j, 9.+0.j])
```

Lưu ý việc chuyển đổi tự động sang định dạng chính xác đã xảy ra.

Chèn chỉ thị vào nguồn Fortran

Giao diện đẹp mắt cũng có thể được tạo tự động bằng cách đặt các chỉ thị biến đổi dưới dạng nhận xét đặc biệt trong mã Fortran gốc. Vì vậy, nếu tôi sửa đổi mã nguồn để chứa:

```
C
    CHƯƠNG TRÌNH CON ZADD(A,B,C,N)
C
Ý ĐỊNH CF2PY(OUT) :: C
Ý ĐỊNH CF2PY(ẨN) :: N
CF2PY PHỨC HỢP NHÂN ĐÔI :: A(N)
TỔ HỢP NHÂN ĐÔI CF2PY :: B(N)
CF2PY PHỨC HỢP NHÂN ĐÔI :: C(N)
    TỔNG HỢP NHÂN ĐÔI A(*)
    PHỨC HỢP NHÂN ĐÔI B(*)
    PHỨC HỢP NHÂN ĐÔI C(*)
    số nguyên N
LÀM 20 J = 1, N
    C(J) = A(J) + B(J)
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

20 TIẾP TỤC

KẾT THÚC

Sau đó, tôi có thể biên dịch môđun mở rộng bằng cách sử dụng:

```
f2py -c -m thêm add.f
```

Chữ ký thu được cho hàm add.zadd hoàn toàn giống với chữ ký đã được tạo trước đó. Nếu mã nguồn ban đầu chứa A(N) thay vì A(\*) v.v. với B và C, thì tôi có thể có được (gần) giao diện tương tự chỉ bằng cách đặt dòng nhận xét INTENT(OUT) :: C trong mã nguồn. Sự khác biệt duy nhất là N sẽ là đầu vào tùy chọn có độ dài mặc định là A.

Ví dụ lọc

Để so sánh với các phương pháp khác sẽ được thảo luận. Đây là một ví dụ khác về hàm lọc một mảng hai chiều gồm các số đầu phẩy động có độ chính xác kép bằng cách sử dụng bộ lọc trung bình cố định. Ưu điểm của việc sử dụng Fortran để lập chỉ mục thành các mảng đa chiều phải rõ ràng từ ví dụ này.

```
SUBROUTINE DFILTER2D(A,B,M,N)
C
    ĐỘ CHÍNH XÁC NHÂN ĐÔI A(M,N)
    ĐỘ CHÍNH XÁC NHÂN ĐÔI B(M,N)
    SỐ N, M
    Ý ĐỊNH CF2PY(OUT) :: B
    Ý ĐỊNH CF2PY(ÂN) :: N
    Ý ĐỊNH CF2PY(ÂM) :: M
    DO 20 I = 2,M-1
        LÀM 40 J=2,N-1
            B(I,J) = A(I,J) +
            ,           (A(I-1,J)+A(I+1,J) +
            ,           A(I,J-1)+A(I,J+1))*0.5D0 +
            (A(I-1,J- 1) + A(I-1,J+1) +
            A(I+1,J-1) + A(I+1,J+1))*0.25D0
40
    TIẾP TỤC
20 TIẾP TỤC
KẾT THÚC
```

Mã này có thể được biên dịch và liên kết thành một môđun mở rộng có tên filter bằng cách sử dụng:

```
f2py -c -m bộ lọc filter.f
```

Điều này sẽ tạo ra một môđun mở rộng có tên filter.so trong thư mục hiện tại với phương thức có tên dfilter2d trả về phiên bản đã lọc của đầu vào.

Gọi f2py từ Python

Chương trình f2py được viết bằng Python và có thể chạy từ bên trong mã của bạn để biên dịch mã Fortran khi chạy, như sau:

```
từ numpy import f2py với
open("add.f") dưới dạng tệp nguồn:
    mã nguồn = sourcefile.read()
f2py.compile(sourcecode, modulename='add') nhập
    thêm
```

Chuỗi nguồn có thể là bất kỳ mã Fortran hợp lệ nào. Nếu bạn muốn lưu mã nguồn mở rộng thì tên tệp phù hợp có thể được cung cấp bằng từ khóa `source_fn` cho hàm biên dịch.

#### Tạo mô-đun mở rộng tự động

Nếu bạn muốn phân phối mô-đun tiện ích mở rộng f2py của mình thì bạn chỉ cần bao gồm tệp .pyf và mã Fortran. Các phần mở rộng distutils trong NumPy cho phép bạn xác định hoàn toàn một mô-đun mở rộng theo tệp giao diện này. Tệp `setup.py` hợp lệ cho phép phân phối mô-đun `add.f` (như một phần của gói `f2py_examples` để nó được tải dưới dạng `f2py_examples.add`) là:

```
cấu hình def (parent_package='', top_path=None)
    từ numpy.distutils.misc_util nhập Cấu hình cấu hình = Cấu hình
    ('f2py_examples',parent_package, top_path) config.add_extension('add', nguồn=['add.pyf', 'add.f'])
    trả về cấu hình

nếu __name__ == '__main__':
    từ thiết lập thiết lập nhập numpy.distutils.core
    (**configuration(top_path='').todict())
```

Việc cài đặt gói mới rất dễ dàng bằng cách sử dụng:

```
cài đặt pip .
```

giả sử bạn có quyền thích hợp để ghi vào thư mục trang web chính cho phiên bản Python bạn đang sử dụng. Để gói kết quả hoạt động, bạn cần tạo một tệp có tên `__init__.py` (trong cùng thư mục với `add.pyf`). Lưu ý mô-đun mở rộng được xác định hoàn toàn theo các tệp `add.pyf` và `add.f`. Việc chuyển đổi tệp .pyf thành tệp .c được xử lý bởi `numpy.distutils`.

#### Phản kết luận

Tệp định nghĩa giao diện (.pyf) là cách bạn có thể tinh chỉnh giao diện giữa Python và Fortran. Có tài liệu phù hợp về f2py được tìm thấy trong thư mục `numpy/f2py/docs` bất cứ khi nào NumPy được cài đặt trên hệ thống của bạn (thường là trong các gói trang web). Ngoài ra còn có thêm thông tin về cách sử dụng f2py (bao gồm cách sử dụng nó để bọc mã C) tại <https://scipy-cookbook.readthedocs.io> dưới tiêu đề "Giao diện với các ngôn ngữ khác".

Phương pháp liên kết mã được biên dịch f2py hiện là phương pháp tích hợp và phức tạp nhất. Nó cho phép phân tách rõ ràng Python bằng mã được biên dịch trong khi vẫn cho phép phân phối riêng mô-đun mở rộng. Hạn chế duy nhất là nó yêu cầu sự tồn tại của trình biên dịch Fortran để người dùng cài đặt mã. Tuy nhiên, với sự tồn tại của các trình biên dịch miễn phí g77, gfortran và g95, cũng như các trình biên dịch thương mại chất lượng cao, hạn chế này không đặc biệt nghiêm trọng. Theo tôi, Fortran vẫn là cách dễ nhất để viết mã nhanh và rõ ràng cho máy tính khoa học. Nó xử lý các số phức và lập chỉ mục đa chiều theo cách đơn giản nhất.

Tuy nhiên, hãy lưu ý rằng một số trình biên dịch Fortran sẽ không thể tối ưu hóa mã tốt như mã C viết tay tốt.

### 7.2.4 Cython

Cython là một trình biên dịch cho phương ngữ Python có thêm kiểu gõ tĩnh (tùy chọn) để tăng tốc độ và cho phép trộn mã C hoặc C++ vào các mô-đun của bạn. Nó tạo ra các phần mở rộng C hoặc C++ có thể được biên dịch và nhập vào mã Python.

Nếu bạn đang viết một mô-đun mở rộng bao gồm khá nhiều mã thuật toán của riêng bạn thì Cython là một lựa chọn phù hợp. Một trong những tính năng của nó là khả năng làm việc dễ dàng và nhanh chóng với các mảng đa chiều.

Lưu ý rằng Cython chỉ là một trình tạo mô-đun mở rộng. Không giống như f2py, nó không có cơ sở tự động để biên dịch và liên kết mô-đun mở rộng (việc này phải được thực hiện theo cách thông thường). Nó cung cấp một lớp distutils đã sửa đổi có tên build\_ext cho phép bạn xây dựng một mô-đun mở rộng từ nguồn .pyx. Vì vậy, bạn có thể viết vào tệp setup.py:

```
từ Cython.Distutils nhập build_ext từ nhập
distutils.extension Tiện ích mở rộng từ distutils.core nhập
thiết lập nhập numpy

setup(name='mine', description='Nothing',
      ext_modules=[Extension('filter', ['filter.pyx'],
                            include_dirs=[numpy.get_include()]),
      cmdclass = {'build_ext':build_ext})
```

Tất nhiên, việc thêm thư mục bao gồm NumPy chỉ cần thiết nếu bạn đang sử dụng mảng NumPy trong mô-đun mở rộng (đó là lý do chúng tôi cho rằng bạn đang sử dụng Cython). Các tiện ích mở rộng distutils trong NumPy cũng bao gồm hỗ trợ tự động tạo mô-đun mở rộng và liên kết nó từ tệp .pyx. Nó hoạt động sao cho nếu người dùng chưa cài đặt Cython thì nó sẽ tìm một tệp có cùng tên tệp nhưng có phần mở rộng .c mà sau đó nó sẽ sử dụng thay vì cố gắng tạo lại tệp .c.

Nếu bạn chỉ sử dụng Cython để biên dịch mô-đun Python tiêu chuẩn thì bạn sẽ nhận được mô-đun mở rộng C thường chạy nhanh hơn một chút so với mô-đun Python tương đương. Có thể tăng tốc độ hơn nữa bằng cách sử dụng từ khóa cdef để xác định tĩnh các biến C.

Hãy xem hai ví dụ chúng ta đã thấy trước đây để biết cách chúng có thể được triển khai bằng Cython. Những ví dụ này được biên soạn thành các mô-đun mở rộng bằng Cython 0.21.1.

#### Phép cộng phức tạp trong Cython

Đây là một phần của mô-đun Cython có tên add.pyx, thực hiện các hàm cộng phức tạp mà chúng tôi đã triển khai trước đây bằng f2py:

```
cimport cython
cimport numpy as np nhập
numpy as np

# Chúng ta cần khởi tạo NumPy. np.import_array()

#@cython.boundscheck(False) def zadd(in1,
in2): cdef double complex[:,]
    a = in1.ravel() cdef double complex[:] b = in2.ravel()

    out = np.empty(a.shape[0], np.complex64) cdef double
    complex[:] c = out.ravel()

    cho tôi trong phạm vi (c.shape[0]):
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
c[i].real = a[i].real + b[i].real c[i].imag
= a[i].imag + b[i].imag
```

trở về

Mô-đun này cho thấy việc sử dụng câu lệnh `cimport` để tải các định nghĩa từ tiêu đề `numpy.pxd` đi kèm với Cython. Có vẻ như NumPy được nhập hai lần; `cimport` chỉ cung cấp NumPy C-API, trong khi quá trình nhập thông thường tạo ra quá trình nhập kiểu Python khi chạy và cho phép gọi vào API NumPy Python quen thuộc.

Ví dụ này cũng minh họa “các chế độ xem bộ nhớ đã nhập” của Cython, giống như mảng NumPy ở cấp độ C, theo nghĩa là chúng là các mảng được định hình và phân cấp biệt phem vi riêng của chúng (không giống như mảng C được xử lý thông qua một con trỏ平凡). Cú pháp `double complex[:]` biểu thị một mảng một chiều (vector) nhân đôi, với các bước tùy ý. Một mảng `int` liền kề sẽ là `int[:, :]`, trong khi một ma trận `float` sẽ là `float[:, :, :]`.

Được nhận xét được hiển thị là trình trang trí `cython.boundscheck`, công cụ này sẽ kiểm tra giới hạn để xem bật hoặc tắt quyền truy cập bộ nhớ trên cơ sở từng chức năng. Chúng ta có thể sử dụng điều này để tăng tốc mã của mình hơn nữa, nhưng lại gây bất lợi cho sự an toàn (hoặc kiểm tra thủ công trước khi vào vòng lặp).

Ngoài cú pháp dạng `xem`, hàm này có thể được đọc ngay lập tức đối với lập trình viên Python. Kiểu gõ tinh của biến `i` là `ắn`. Thay vì cú pháp dạng `xem`, chúng ta cũng có thể sử dụng cú pháp mảng NumPy đặc biệt của Cython, nhưng cú pháp dạng `xem` được ưu tiên hơn.

#### Bộ lọc hình ảnh trong Cython

Ví dụ hai chiều mà chúng tôi đã tạo bằng Fortran cũng dễ viết bằng Cython:

```
cimport numpy as np
np import_array()

bộ lọc def(img):
    cdef double[:, :] a = np.asarray(img, dtype=np.double) out =
    np.zeros(img.shape, dtype=np.double) cdef double[:, ::
    1] b = ra ngoài

    cdef np.npy_intp i, j

    cho i trong phạm vi (1, a.shape [0] - 1):
        cho j trong phạm vi(1, a.shape[1] - 1): b[i,
            j] = (a[i, j]
                + .5 * ( a[i-1, j] + a[i+1, j] + a[i,
                    j-1] + a[i, j+1]) + .25 *
                ( a[i- 1, j-1] + a[i-1, j+1] + a[i+1, j-1] +
                    a[i+1, j+1]))
```

trở về

Bộ lọc trung bình 2-d này chạy nhanh vì vòng lặp nằm trong C và việc tính toán con trỏ chỉ được thực hiện khi cần thiết.

Nếu đoạn mã trên được biên dịch dưới dạng hình ảnh mô-đun thì hình ảnh 2 chiều, `img`, có thể được lọc rất nhanh bằng cách sử dụng:

```
nhập hình ảnh
ra = image.filter(img)
```

Về mã, có hai điều cần lưu ý: thứ nhất, không thể trả lại chế độ xem bộ nhớ cho Python. Thay vào đó, mảng NumPy out được tạo trước tiên, sau đó chế độ xem b trên mảng này được sử dụng để tính toán. Thứ hai, view b được gõ `double[:, ::1]`. Điều này có nghĩa là mảng 2 chiều với các hàng liền kề, tức là thứ tự ma trận C. Việc chỉ định thứ tự một cách rõ ràng có thể tăng tốc một số thuật toán vì chúng có thể bỏ qua các bước tính toán.

#### Phản kết luận

Cython là cơ chế mở rộng được lựa chọn cho một số thư viện Python khoa học, bao gồm Scipy, Pandas, SAGE, scikit-image và scikit-learn, cũng như thư viện xử lý XML LXML. Ngôn ngữ và trình biên dịch được duy trì tốt.

Có một số nhược điểm khi sử dụng Cython:

1. Khi mã hóa các thuật toán tùy chỉnh và đôi khi khi gói các thư viện C hiện có, cần phải làm quen với C. Đặc biệt, khi sử dụng quản lý bộ nhớ C (`malloc` và `free`), rất dễ gây rò rỉ bộ nhớ. Tuy nhiên, chỉ cần biên dịch một mô-đun Python được đổi tên thành `.pyx` là có thể tăng tốc nó và việc thêm một vài khai báo kiểu có thể giúp tăng tốc đáng kể trong một số mã.
2. Rất dễ mất đi sự tách biệt rõ ràng giữa Python và C, điều này khiến cho việc sử dụng lại mã C của bạn cho các ngôn ngữ khác phải Python khác trở nên dễ dàng hơn. Các dự án liên quan đến Python khó khăn hơn.
3. Mã C do Cython tạo ra rất khó đọc và sửa đổi (và thường được biên dịch với các lỗi khó chịu nhưng vô hại). cảnh báo).

Một lợi thế lớn của các mô-đun mở rộng do Cython tạo là chúng dễ phân phối. Tóm lại, Cython là một công cụ rất có khả năng để dán mã C hoặc tạo mô-đun mở rộng một cách nhanh chóng và không nên bỏ qua.

Nó đặc biệt hữu ích cho những người không thể hoặc sẽ không viết mã C hoặc Fortran.

#### 7.2.5 loại c

`ctypes` là một mô-đun mở rộng Python, có trong `stdlib`, cho phép bạn gọi một hàm tùy ý trong thư viện dùng chung trực tiếp từ Python. Cách tiếp cận này cho phép bạn giao tiếp với mã C trực tiếp từ Python. Điều này mở ra một số lượng lớn các thư viện để sử dụng từ Python. Tuy nhiên, nhược điểm là các lỗi mã hóa có thể dẫn đến chương trình bị treo rất dễ dàng (giống như có thể xảy ra trong C) vì có rất ít việc kiểm tra loại giới hạn được thực hiện trên các tham số. Điều này đặc biệt đúng khi dữ liệu mảng được truyền vào dưới dạng con trỏ tới vị trí bộ nhớ thô. Khi đó, trách nhiệm thuộc về bạn là chương trình con sẽ không truy cập vào bộ nhớ bên ngoài vùng mảng thực tế. Tuy nhiên, nếu bạn không ngại việc sử dụng một chút nguy hiểm thì `ctypes` có thể là một công cụ hiệu quả để nhanh chóng tận dụng một thư viện dùng chung lớn (hoặc viết chức năng mở rộng trong thư viện dùng chung của riêng bạn).

Bởi vì cách tiếp cận `ctypes` hiển thị giao diện thô cho mã được biên dịch nên không phải lúc nào cũng chấp nhận được lỗi của người dùng. Việc sử dụng mô-đun `ctypes` một cách hiệu quả thường bao gồm một lớp mã Python bổ sung để kiểm tra các kiểu dữ liệu và giới hạn mảng của các đối tượng được chuyển đến chương trình con cơ bản. Lớp kiểm tra bổ sung này (chưa kể đến việc chuyển đổi từ đối tượng `ctypes` sang loại dữ liệu C mà `ctypes` tự thực hiện), sẽ làm cho giao diện chậm hơn so với giao diện mô-đun mở rộng viết tay. Tuy nhiên, chi phí này sẽ không đáng kể nếu quy trình C được gọi đang thực hiện bất kỳ khởi lượng công việc đáng kể nào. Nếu bạn là một lập trình viên Python giỏi với kỹ năng C yếu, `ctypes` là một cách dễ dàng để viết giao diện hữu ích cho thư viện mã được biên dịch (được chia sẻ).

#### Để sử dụng `ctypes` bạn phải

1. Có thư viện dùng chung.
2. Tải thư viện dùng chung.
3. Chuyển đổi các đối tượng python thành các đối số được hiểu bằng `ctypes`.
4. Gọi hàm từ thư viện với các đối số `ctypes`.

## Có thư viện dùng chung

Có một số yêu cầu đối với thư viện dùng chung có thể được sử dụng với các loại ctype dành riêng cho nền tảng. Hướng dẫn này giả định rằng bạn đã quen với việc tạo thư viện dùng chung trên hệ thống của mình (hoặc đơn giản là có sẵn thư viện dùng chung cho bạn). Các mục cần nhớ là:

- Thư viện dùng chung phải được biên dịch theo cách đặc biệt (ví dụ: sử dụng cờ -shared với gcc).
- Trên một số nền tảng (ví dụ Windows), thư viện dùng chung yêu cầu tệp .def chỉ định các chức năng được xuất. Ví dụ: tệp mylib.def có thể chứa:

```
THƯ VIỆN mylib.dll XUẤT KHẨU
cool_function1
cool_function2
```

Ngoài ra, bạn có thể sử dụng bộ xác định lớp lưu trữ \_\_declspec(dllexport) trong định nghĩa C của hàm để tránh sự cần thiết của tệp .def này.

Không có cách tiêu chuẩn nào trong Python distutils để tạo một thư viện chia sẻ tiêu chuẩn (mô-đun mở rộng là thư viện chia sẻ “đặc biệt” mà Python hiểu) theo cách đa nền tảng. Do đó, nhược điểm lớn của ctypes tại thời điểm viết cuốn sách này là khó phân phối đa nền tảng một tiện ích mở rộng Python sử dụng ctypes và bao gồm mã của riêng bạn. Mã này sẽ được biên dịch thành thư viện dùng chung trên hệ thống người dùng. . .

## Đang tải thư viện chia sẻ

Một cách đơn giản nhưng mạnh mẽ để tải thư viện dùng chung là lấy tên đường dẫn tuyệt đối và tải nó bằng đối tượng cdll của ctypes:

```
lib = ctypes.cdll[<full_path_name>]
```

Tuy nhiên, trên Windows việc truy cập một thuộc tính của phương thức cdll sẽ tải DLL đầu tiên có tên đó được tìm thấy trong thư mục hiện tại hoặc trên PATH. Việc tải tên đường dẫn tuyệt đối đòi hỏi một chút khéo léo để làm việc trên nhiều nền tảng do phần mở rộng của các thư viện dùng chung khác nhau. Có sẵn một tiện ích ctypes.util.find\_library có thể đơn giản hóa quá trình tìm thư viện để tải nhưng nó không phải là dễ dàng. Vẫn đề phức tạp hơn, các nền tảng khác nhau có các phần mở rộng mặc định khác nhau được các thư viện chia sẻ sử dụng (ví dụ: .dll - Windows, .so - Linux, .dylib - Mac OS X). Điều này cũng phải được tính đến nếu bạn đang sử dụng ctypes để bọc mã cần hoạt động trên nhiều nền tảng.

NumPy cung cấp một hàm tiện lợi gọi là `ctypeslib.load_library` (tên, đường dẫn). Hàm này lấy tên của thư viện dùng chung (bao gồm mọi tiền tố như 'lib' nhưng không bao gồm phần mở rộng) và đường dẫn nơi có thể đặt thư viện dùng chung. Nó trả về một đối tượng thư viện ctypes hoặc tăng OSError nếu không thể tìm thấy thư viện hoặc tăng ImportError nếu mô-đun ctypes không có sẵn. (Người dùng Windows: đối tượng thư viện ctypes được tải bằng `Load_library` luôn được tải giả sử quy ước gọi `cdecl`. Xem tài liệu ctypes trong `ctypes.windll` và/hoặc `ctypesoledll` để biết cách tải thư viện theo các quy ước gọi khác).

Các hàm trong thư viện dùng chung có sẵn dưới dạng thuộc tính của đối tượng thư viện ctypes (được trả về từ `ctypeslib.Load_library`) hoặc dưới dạng các mục sử dụng cú pháp `lib['func_name']`. Phương pháp thứ hai để truy xuất tên hàm đặc biệt hữu ích nếu tên hàm chứa các ký tự không được phép trong tên biến Python.

## chuyển đổi đối số

Các đối tượng int/long, chuỗi và unicode trong Python được tự động chuyển đổi khi cần thành các đối số ctypes tương đương. Đối tượng None cũng được tự động chuyển đổi thành con trỏ NULL. Tất cả các đối tượng Python khác phải được chuyển đổi thành các loại dành riêng cho ctypes. Có hai cách xung quanh hạn chế này cho phép ctypes tích hợp với các đối tượng khác.

1. Không đặt thuộc tính `argtypes` của đối tượng hàm và xác định phương thức `_as_parameter_` cho đối tượng bạn muốn truyền vào. Phương thức `_as_parameter_` phải trả về một int Python sẽ được truyền trực tiếp vào hàm.
2. Đặt thuộc tính `argtypes` thành một danh sách có các mục chứa các đối tượng có phương thức lớp có tên `from_param` biết cách chuyển đổi đối tượng của bạn thành đối tượng mà ctypes có thể hiểu được (int/long, string, unicode hoặc đối tượng có thuộc tính `_as_parameter_`).

NumPy sử dụng cả hai phương pháp với ưu tiên cho phương pháp thứ hai vì nó có thể an toàn hơn. Thuộc tính `ctypes` của `ndarray` trả về một đối tượng có thuộc tính `_as_parameter_` trả về một số nguyên biểu thị địa chỉ của `ndarray` mà nó được liên kết. Kết quả là, người ta có thể truyền trực tiếp đối tượng thuộc tính `ctypes` này tới một hàm mong đợi một con trỏ tới dữ liệu trong `ndarray` của bạn. Người gọi phải chắc chắn rằng đối tượng `ndarray` có đúng loại, hình dạng và có các cờ chính xác được đặt, nếu không sẽ có nguy cơ gặp sự cố khó chịu nếu con trỏ dữ liệu tới các mảng không phù hợp được truyền vào.

Để triển khai phương thức thứ hai, NumPy cung cấp hàm `ndpointer` của lớp nhà máy trong tệp `numpy.mro`. mô-đun `ctypeslib`. Hàm nhà máy lớp này tạo ra một lớp thích hợp có thể được đặt trong mục nhập thuộc tính `argtypes` của hàm `ctypes`. Lớp này sẽ chứa một phương thức `from_param` mà ctypes sẽ sử dụng để chuyển đổi bất kỳ `ndarray` nào được truyền vào hàm thành một đối tượng được ctypes nhận dạng. Trong quá trình này, quá trình chuyển đổi sẽ thực hiện kiểm tra bất kỳ thuộc tính nào của `ndarray` được người dùng chỉ định trong lệnh gọi tới `ndpointer`. Các khía cạnh của `ndarray` có thể được kiểm tra bao gồm kiểu dữ liệu, số thứ nguyên, hình dạng và/hoặc trạng thái của các cờ trên bất kỳ mảng nào được truyền. Giá trị trả về của phương thức `from_param` là thuộc tính `ctypes` của mảng (vì nó chưa thuộc tính `_as_parameter_` trả đến vùng dữ liệu mảng) có thể được ctypes sử dụng trực tiếp.

Thuộc tính `ctypes` của `ndarray` cũng được trang bị các thuộc tính bổ sung có thể thuận tiện khi chuyển thông tin bổ sung về mảng vào hàm `ctypes`. Dữ liệu thuộc tính, hình dạng và bước tiến có thể cung cấp các loại tương thích với ctypes tương ứng với vùng dữ liệu, hình dạng và bước tiến của mảng. Thuộc tính dữ liệu trả về `c_void_p` đại diện cho một con trỏ tới vùng dữ liệu. Mỗi thuộc tính hình dạng và sải bước trả về một mảng số nguyên ctypes (hoặc Không biểu thị con trỏ NULL, nếu mảng 0-d). Ctype cơ sở của mảng là một số nguyên ctype có cùng kích thước với một con trỏ trên nền tảng. Ngoài ra còn có các phương thức `data_as({ctype})`, `shape_as(<base ctype>)` và sải bước `as(<base ctype>)`. Chúng trả về dữ liệu dưới dạng đối tượng ctype mà bạn chọn và các mảng hình dạng/bước tiến sử dụng loại cơ sở cơ bản mà bạn chọn. Để thuận tiện, mô-đun `ctypeslib` cũng chứa `c_intp` dưới dạng kiểu dữ liệu số nguyên ctypes có kích thước giống với kích thước của `c_void_p` trên nền tảng (giá trị của nó là Không nếu ctypes không được cài đặt).

## Gọi hàm

Hàm được truy cập dưới dạng thuộc tính hoặc mục từ thư viện chia sẻ đã tải. Do đó, nếu `./mylib.so` có hàm có tên `cool_function1`, Tôi có thể truy cập chức năng này dưới dạng:

```
lib = numpy.ctypeslib.load_library('mylib','.')
func1 = lib.cool_function1 # hoặc tương đương func1 =
lib['cool_function1']
```

Trong ctypes, giá trị trả về của hàm được đặt thành 'int' theo mặc định. Hành vi này có thể được thay đổi bằng cách đặt thuộc tính `restype` của hàm. Sử dụng Không có để gõ lại nếu hàm không có giá trị trả về ('void'):

```
func1.restype = Không có
```

Như đã thảo luận trước đó, bạn cũng có thể đặt thuộc tính `argtypes` của hàm để ctypes kiểm tra loại đối số đầu vào khi hàm được gọi. Sử dụng hàm xuất xưởng `ndpointer` để tạo một lớp làm sẵn để kiểm tra kiểu dữ liệu, hình dạng và cờ trên hàm mới của bạn. Hàm `ndpointer` có chữ ký

```
ndpointer(dtype=Không có, ndim=Không có, hình dạng=Không có, flags=Không có)
```

Đối số từ khóa có giá trị Không có không được chọn. Việc chỉ định một từ khóa sẽ thực thi việc kiểm tra khía cạnh đó của ndarray khi chuyển đổi sang đối tượng tương thích với ctypes. Từ khóa dtype có thể là bất kỳ đối tượng nào được hiểu là đối tượng kiểu dữ liệu. Từ khóa ndim phải là số nguyên và từ khóa hình dạng phải là số nguyên hoặc một dãy số nguyên. Từ khóa flags chỉ định các cờ tối thiểu được yêu cầu trên bất kỳ mảng nào được truyền vào. Điều này có thể được chỉ định dưới dạng một chuỗi các yêu cầu được phân tách bằng dấu phẩy, một số nguyên biểu thị các bit yêu cầu OR cùng nhau hoặc một đối tượng cờ được trả về từ thuộc tính cờ của một mảng có các yêu cầu cần thiết.

Việc sử dụng lớp ndpointer trong phương thức argtypes có thể giúp việc gọi hàm C bằng cách sử dụng ctypes và vùng dữ liệu của ndarray trở nên an toàn hơn đáng kể. Bạn vẫn có thể muốn bọc hàm trong một trình bao bọc Python bổ sung để làm cho nó thân thiện với người dùng (ẩn một số đối số rõ ràng và tạo một số đối số đầu ra đối số). Trong quá trình này, hàm yêu cầu trong NumPy có thể hữu ích để trả về đúng loại mảng từ một đầu vào nhất định.

#### Ví dụ hoàn chỉnh

Trong ví dụ này, tôi sẽ chỉ ra cách triển khai hàm cộng và hàm lọc được triển khai trước đó bằng cách sử dụng ctypes. Đầu tiên, mã C thực hiện các thuật toán chứa các hàm zadd, Dadd, sadd, cadd và dfilter2d. Hàm zadd là:

```
/* Thêm các mảng dữ liệu liền kề */ typedef struct
{double real; hình ảnh kép;} cdouble; typedef struct {float thực; hình ảnh
nối;} cfloating; void zadd(cdouble *a, cdouble *b, cdouble *c, long n) {

    trong khi (n--) {
        c->real = a->real + b->real; c-
        >hình ảnh = a->hình ảnh + b->hình
        ảnh; một++; b++; C++;
    }
}
```

với mã tương tự cho cadd, Dadd và Sadd xử lý các kiểu dữ liệu float, double và float phức tạp tương ứng:

```
void cadd(cfloating *a, cfloating *b, cfloating *c, long n) {

    trong khi (n--) {
        c->real = a->real + b->real; c-
        >hình ảnh = a->hình ảnh + b->hình
        ảnh; một++; b++; C++;
    }
}

void Dadd(double *a, double *b, double *c, long n) {

    trong khi (n--) {
        *c++ = *a++ + *b++;
    }
}

void sadd(float *a, float *b, float *c, long n) {

    while (n--) { *c+
        + = *a++ + *b++;
    }
}
```

Tệp code.c cũng chứa hàm dfilter2d:

```

'
 * Giả sử b liền kề và có các bước là bội số của sizeof(double) */

trong
dfilter2d(double *a, double *b, ssize_t *astrides, ssize_t *dims) {

    ssize_t i, j, M, N, S0, S1;
    ssize_t r, c, rm1, rp1, cp1, cm1;

    M = mờ[0]; N = độ mờ[1];
    S0 = bước đi[0]/sizeof(double);
    S1 = bước tiến [1]/sizeof(double); vì (i = 1;
    i < M - 1; i++) {
        r = i*S0;
        rp1 = r + S0;
        rm1 = r - S0;
        cho (j = 1; j < N - 1; j++) {
            c = j*S1;
            cp1 = j + S1;
            cm1 = j - S1;
            b[i*N + j] = a[r + c] +
                (a[rp1 + c] + a[rm1 + c] +
                a[r + cp1] + a[r + cm1])*0.5 +
                (a[rp1 + cp1] + a[rp1 + cm1] +
                a[rm1 + cp1] + a[rm1 + cm1])*0.25;
        }
    }
}

```

Một lợi thế có thể có của mã này so với mã tương đương với Fortran là nó có bước tiến tùy ý (tức là các mảng không liền kề) và cũng có thể chạy nhanh hơn tùy thuộc vào khả năng tối ưu hóa của trình biên dịch của bạn. Tuy nhiên, nó rõ ràng là phức tạp hơn mã đơn giản trong filter.f. Mã này phải được biên dịch thành thư viện dùng chung.

Trên hệ thống Linux của tôi, việc này được thực hiện bằng cách sử dụng:

```
gcc -o code.so -shared code.c
```

Việc này tạo ra một thư viện chia sẻ có tên code.so trong thư mục hiện tại. Trên Windows, đừng quên thêm \_\_declspec(dllexport) vào trước khoảng trống trên dòng trước mỗi định nghĩa hàm hoặc viết tệp code.def liệt kê tên của các hàm sẽ được xuất.

Nên xây dựng giao diện Python phù hợp cho thư viện dùng chung này. Để thực hiện việc này, hãy tạo một tệp có tên Interface.py với các dòng sau ở trên cùng:

```

__all__ = ['thêm', 'filter2d']

nhập numpy dưới dạng
np nhập os

_path = os.path.dirname('__file__') lib =
np.ctypeslib.load_library('code', _path) _typedict =
{'zadd' : complex, 'sadd' : np.single, 'cadd' : np.csingle ,
'dadd' : float} cho tên trong _typedict.keys():
val = getattr(lib, name) val.restype
= None _type = _typedict[name]

```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
val.argtypes = [np.ctypeslib.ndpointer(_type,
                                         flags='căn chỉnh, liền kề'),
                np.ctypeslib.ndpointer(_type,
                                         flags='căn chỉnh, liền kề'),
                np.ctypeslib.ndpointer(_type,
                                         flags='căn chỉnh, liền kề, \'\
                                         có thể ghi'),
                np.ctypeslib.c_intp]
```

Mã này tải thư viện dùng chung có tên code.{ext} nằm trong cùng đường dẫn với tệp này. Sau đó, nó thêm kiểu trả về void vào các hàm có trong thư viện. Nó cũng bổ sung tính năng kiểm tra đối số cho các hàm trong thư viện để có thể truyền ndarray dưới dạng ba đối số đầu tiên cùng với một số nguyên (đủ lớn để giữ một con trỏ trên nền tảng) làm đối số thứ tư.

Việc thiết lập hàm lọc cũng tương tự và cho phép gọi hàm lọc với các đối số ndarray làm hai đối số đầu tiên và với các con trỏ tới số nguyên (đủ lớn để xử lý các bước và hình dạng của ndarray) làm hai đối số cuối cùng.:

```
lib.dfilter2d.restype=Không có
lib.dfilter2d.argtypes = [np.ctypeslib.ndpointer(float, ndim=2, flags='aligned'),
                         np.ctypeslib.ndpointer(float, ndim=2,
                                         flags='căn chỉnh, liền kề, \'\
                                         có thể ghi'),
                         ctypes.POINTER(np.ctypeslib.c_intp),
                         ctypes.POINTER(np.ctypeslib.c_intp)]
```

Tiếp theo, xác định một hàm chọn đơn giản để chọn hàm cộng nào sẽ gọi trong thư viện dùng chung dựa trên kiểu dữ liệu:

```
def select(dtype): if
    dtype.char in ['?bBhHf']: trả về lib.sadd, đơn

    elif dtype.char trong ['F']:
        trả về lib.cadd,csingle
    elif dtype.char trong ['DG']:
        trả về lib.zadd, phức tạp
    nếu không thì:
        trả về lib.dadd, float trả về func,
    ntype
```

Cuối cùng, hai hàm được giao diện xuất có thể được viết đơn giản là:

```
def thêm (a, b):
    yêu cầu = ['CONTIGUOUS', 'ALIGNED'] a = np.asanyarray(a)
    func, dtype = select(a.dtype) a
    = np.require(a, dtype, require) b = np.require(b,
    dtype , yêu cầu) c = np.empty_like(a) func(a,b,c,a.size)

    trả lại c
```

Và:

```
def filter2d(a): a =
    np.require(a, float, ['ALIGNED']) b =
    np.zeros_like(a)
    lib.dfilter2d(a, b, a.ctypes.strides, a.ctypes.shape) trả lại b
```

### Phản kết luận

Sử dụng ctypes là một cách mạnh mẽ để kết nối Python với mã C tùy ý. Ưu điểm của nó để mở rộng Python bao gồm

- tách biệt rõ ràng mã C khỏi mã Python
  - không cần học cú pháp mới ngoại trừ Python và C
  - cho phép tái sử dụng mã C
  - chức năng trong các thư viện dùng chung được viết cho các mục đích khác có thể được lấy bằng trình bao bọc Python đơn giản và tìm kiếm thư viện.
- ' tích hợp dễ dàng với NumPy thông qua thuộc tính ctypes
- kiểm tra đối số đầy đủ với nhà máy lớp ndpointer

Nhược điểm của nó bao gồm

- Rất khó để phân phối mô-đun mở rộng được tạo bằng ctypes vì thiếu hỗ trợ xây dựng thư viện được chia sẻ trong distutils (nhưng tôi nghĩ điều này sẽ thay đổi theo thời gian).
- Bạn phải có thư viện chia sẻ mã của mình (không có thư viện tĩnh).
- Rất ít hỗ trợ cho mã C++ và các quy ước gọi thư viện khác nhau của nó. Bạn có thể sẽ cần một C trình bao bọc xung quanh mã C++ để sử dụng với ctypes (hoặc chỉ sử dụng Boost.Python thay thế).

Do khó khăn trong việc phân phối mô-đun mở rộng được tạo bằng ctypes, f2py và Cython vẫn là những cách dễ nhất để mở rộng Python cho việc tạo gói. Tuy nhiên, ctypes trong một số trường hợp là sự thay thế hữu ích. Điều này sẽ mang lại nhiều tính năng hơn cho ctypes, giúp loại bỏ khó khăn trong việc mở rộng Python và phân phối tiện ích mở rộng bằng ctypes.

### 7.2.6 Các công cụ bổ sung mà bạn có thể thấy hữu ích

Những công cụ này được những người khác sử dụng Python nhận thấy là hữu ích nên được đưa vào đây. Chúng được thảo luận riêng vì chúng là những cách cũ hơn để thực hiện mọi việc hiện được xử lý bởi f2py, Cython hoặc ctypes (SWIG, PyFort) hoặc vì tôi không biết nhiều về chúng (SIP, Boost). Tôi chưa thêm liên kết vào các phương pháp này vì kinh nghiệm của tôi là bạn có thể tìm thấy liên kết phù hợp nhất nhanh hơn bằng cách sử dụng Google hoặc một số công cụ tìm kiếm khác và mọi liên kết được cung cấp ở đây sẽ nhanh chóng bị lỗi thời. Đừng cho rằng chỉ vì nó được đưa vào danh sách này nên tôi không nghĩ nó xứng đáng với sự quan tâm của bạn. Tôi bao gồm thông tin về các gói này vì nhiều người nhận thấy chúng hữu ích và tôi muốn cung cấp cho bạn càng nhiều tùy chọn càng tốt để giải quyết vấn đề dễ dàng tích hợp mã của bạn.

## VÒI

Trình tạo giao diện và trình bao bọc đơn giản hóa (SWIG) là một phương pháp cũ và khá ổn định để gói các thư viện C/C++ sang nhiều ngôn ngữ khác. Nó không hiểu cụ thể về mảng NumPy nhưng có thể sử dụng được với NumPy thông qua việc sử dụng các bản đồ kiểu chữ. Có một số sơ đồ kiểu mẫu trong thư mục `numpy/tools/swig` trong `numpy.i` cùng với một mô-đun ví dụ sử dụng chúng. SWIG vượt trội trong việc gói các thư viện C/C++ lớn vì nó có thể (gần như) phân tích cú pháp các tiêu đề của chúng và tự động tạo giao diện. Về mặt kỹ thuật, bạn cần tạo tệp `.i` xác định giao diện. Tuy nhiên, thông thường, tệp `.i` này có thể là một phần của tiêu đề. Giao diện thường cần một chút tinh chỉnh để trở nên rất hữu ích. Khả năng phân tích cú pháp các tiêu đề C/C++ và tự động tạo giao diện này vẫn khiến SWIG trở thành một cách tiếp cận hữu ích để thêm chức năng từ C/C++ vào Python, bất chấp các phương pháp khác đã xuất hiện nhằm mục tiêu nhiều hơn vào Python. SWIG thực sự có thể nhằm mục tiêu các tiện ích mở rộng cho một số ngôn ngữ, nhưng các bản đồ kiểu chữ thường phải dành riêng cho từng ngôn ngữ. Tuy nhiên, với những sửa đổi đối với các sơ đồ kiểu dành riêng cho Python, SWIG có thể được sử dụng để tạo giao diện cho thư viện với các ngôn ngữ khác như Perl, Tcl và Ruby.

Trải nghiệm của tôi với SWIG nhìn chung là tích cực vì nó tương đối dễ sử dụng và khá mạnh mẽ. Tôi đã từng sử dụng nó khá thường xuyên trước khi trở nên thành thạo hơn trong việc viết phần mở rộng C. Tuy nhiên, tôi gặp khó khăn khi viết giao diện tùy chỉnh bằng SWIG vì nó phải được thực hiện bằng cách sử dụng khái niệm sơ đồ kiểu chữ không dành riêng cho Python và được viết bằng cú pháp giống C. Do đó, tôi có xu hướng thích các chiến lược dán khác nhau và sẽ chỉ cố gắng sử dụng SWIG để bao bọc một thư viện C/C++ rất lớn. Tuy nhiên, có những người khác sử dụng SWIG khá vui vẻ.

một hộp

SIP là một công cụ khác để gói các thư viện C/C++ dành riêng cho Python và thường như hỗ trợ rất tốt cho C++. Riverbank Computing đã phát triển SIP để tạo các liên kết Python với thư viện QT. Một tệp giao diện phải được ghi để tạo liên kết, nhưng tệp giao diện trông rất giống tệp tiêu đề C/C++. Mặc dù SIP không phải là trình phân tích cú pháp C++ đầy đủ nhưng nó hiểu khá nhiều cú pháp C++ cũng như các chi tiết đặc biệt của riêng nó cho phép sửa đổi cách thực hiện liên kết Python. Nó cũng cho phép người dùng xác định ánh xạ giữa các kiểu Python và các cấu trúc và lớp C/C++.

## Tăng cường Python

Boost là kho lưu trữ các thư viện C++ và Boost.Python là một trong những thư viện cung cấp giao diện ngắn gọn để liên kết các lớp và hàm C++ với Python. Phần tuyệt vời của phương pháp Boost.Python là nó hoạt động hoàn toàn bằng C++ thuận tiện mà không đưa ra cú pháp mới. Nhiều người dùng C++ báo cáo rằng Boost.Python có thể kết hợp những gì tốt nhất của cả hai thế giới một cách liền mạch. Tôi chưa sử dụng Boost.Python vì tôi không phải là người dùng nhiều C++ và việc sử dụng Boost để gói các chương trình con C đơn giản thường là quá mức cần thiết. Mục đích chính của nó là cung cấp các lớp C++ bằng Python. Vì vậy, nếu bạn có một tập hợp các lớp C++ cần được tích hợp hoàn toàn vào Python, hãy cân nhắc việc tìm hiểu và sử dụng Boost.Python.

## PyFort

PyFort là một công cụ tuyệt vời để gói mã C giống Fortran và Fortran vào Python với sự hỗ trợ cho mảng Số. Nó được viết bởi Paul Dubois, một nhà khoa học máy tính nổi tiếng và là người duy trì đầu tiên của Numeric (hiện đã nghỉ hưu). Điều đáng nói với hy vọng rằng ai đó sẽ cập nhật PyFort để hoạt động với các mảng NumPy hiện hỗ trợ các mảng liền kề kiểu Fortran hoặc kiểu C.

## 7.3 Viết ufunc của riêng bạn

Tôi có sức mạnh!

- Anh ta là dân ông

### 7.3.1 Tạo một hàm phổ quát mới

Trước khi đọc phần này, bạn có thể tự làm quen với những kiến thức cơ bản về tiện ích mở rộng C cho Python bằng cách đọc/đọc lượt các hướng dẫn trong Phần 1 của [Mở rộng và Nhúng Trình thông dịch Python](#), và trong [Cách mở rộng NumPy](#).

Mô-đun umath là mô-đun C do máy tính tạo ra, tạo ra nhiều ufunc. Nó cung cấp rất nhiều ví dụ về cách tạo ra một hàm phổ quát. Việc tạo ufunc của riêng bạn để sử dụng máy ufunc cũng không khó. Giả sử bạn có một hàm mà bạn muốn vận hành từng phần tử trên đầu vào của nó. Bằng cách tạo một ufunc mới, bạn sẽ có được một hàm xử lý

- phát thanh truyền hình
- Vòng lặp N chiều
- chuyển đổi loại tự động với mức sử dụng bộ nhớ tối thiểu
- mang đầu ra tùy chọn

Không khó để tạo ufunc của riêng bạn. Tất cả những gì cần thiết là vòng lặp 1-d cho từng loại dữ liệu bạn muốn hỗ trợ.

Mỗi vòng lặp 1-d phải có một chữ ký cụ thể và chỉ có thể sử dụng ufuncs cho các kiểu dữ liệu có kích thước cố định. Lệnh gọi hàm được sử dụng để tạo một ufunc mới hoạt động trên các kiểu dữ liệu tích hợp được đưa ra bên dưới. Một cơ chế khác được sử dụng để đăng ký ufuncs cho các kiểu dữ liệu do người dùng xác định.

Trong một số phần tiếp theo, chúng tôi sẽ đưa ra mã ví dụ có thể dễ dàng sửa đổi để tạo ufuncs của riêng bạn. Các ví dụ lần lượt là các phiên bản đầy đủ hơn hoặc phức tạp hơn của hàm logit, một hàm phổ biến trong mô hình thống kê. Logit cũng thú vị bởi vì, do sự kỳ diệu của các tiêu chuẩn IEEE (cụ thể là IEEE 754), tất cả các hàm logit được tạo bên dưới đều tự động có hành vi sau.

```
>>> logit(0)
-inf
>>> logit(1)
inf
>>> logit(2)
nan
>>> logit(-2)
nan
```

Điều này thật tuyệt vời vì người viết hàm không phải truyền inf hoặc nans theo cách thủ công.

### 7.3.2 Ví dụ về phần mở rộng không phải ufunc

Để so sánh và giúp người đọc hiểu rõ hơn, chúng tôi cung cấp một cách triển khai đơn giản phần mở rộng C của logit không sử dụng numpy.

Để làm điều này, chúng ta cần hai tập tin. Đầu tiên là tập C chứa mã thực tế và thứ hai là tập setup.py được sử dụng để tạo mô-đun.

```
#include <Python.h>
#include <math.h>
```

```
'
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```

* spammodule.c * Đây
là mã C cho phần mở rộng Python không gọn gàng để * xác định hàm logit, trong đó
logit(p) = log(p/(1-p)).
* Chức năng này sẽ không tự động hoạt động trên các mảng có nhiều mảng. * numpy.vectorize phải
được gọi trong python để tạo * một hàm thân thiện với numpy.

,
* Bạn có thể tìm thấy thông tin chi tiết giải thích về API Python-C trong * 'Mở rộng
và nhúng' và 'API Python/C' tại * docs.python.org. ,

/* Cái này khai báo hàm logit */ static
PyObject* spam_logit(PyObject *self, PyObject *args);

/*
* Điều này cho Python biết mô-đun này có những phương thức nào.
* Xem API Python-C để biết thêm thông tin. ,
PyMethodDef SpamMethods tinh [] = { {"logit",
spam_logit,

    METH_VARARGS, "tính nhật ký"},
{NULL, KHONG, 0, KHONG}
,

/*
* Điều này thực sự xác định hàm logit cho * đối số đầu vào từ Python. ,
tinh PyObject* spam_logit(PyObject *self, PyObject *args) {

    đôi p;

    /* Điều này phân tích đối số Python thành một double */ if(!
PyArg_ParseTuple(args, "d", &p)) {
        trả về NULL;
    ,

    /* CHỨC NĂNG LOGIT THỰC TẾ */ p = p/(1-p); p = log(p);

    /*Điều này xây dựng câu trả lời trả lại thành một đối tượng python */ return
Py_BuildValue("d", p);
}

/*
* Thao tác này sẽ khởi tạo mô-đun bằng cách sử dụng các định nghĩa ở trên. */ #if
PY_VERSION_HEX >= 0x03000000 câu trúc tinh
PyModuleDef moduledef = { PyModuleDef_HEAD_INIT, "spam",

```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```

vô giá
trị, -1,
Phương thức thư rác,
vô GIÁ TRỊ,
vô GIÁ TRỊ,
vô GIÁ TRỊ,
vô GIÁ TRỊ

'

PyMODINIT_FUNC PyInit_spam(void) {

    PyObject *m;
    PyModule_Create(&moduledef); nếu (! m) {

        trả về NULL;
    }

    trả lại m;

} #khác

PyMODINIT_FUNC initspam(void) {

    PyObject *m;

    m = Py_InitModule("spam", SpamMethods); if (m == NULL) { trả
    về;

}

}
#endif

```

Để sử dụng tệp setup.py, hãy đặt setup.py và spammodule.c vào cùng một thư mục. Sau đó, python setup.py build sẽ xây dựng mô-đun để nhập hoặc setup.py install sẽ cài đặt mô-đun vào thư mục gói trang web của bạn.

```

'
tập tin setup.py cho spammodule.c

Việc gọi
$python setup.py build_ext --inplace sẽ xây
dựng thư viện tiện ích mở rộng trong tệp hiện tại.

Việc gọi
$python setup.py build sẽ xây
dựng một tệp trông giống như ./build/lib*, trong đó lib* là tệp bắt đầu bằng lib.
Thư viện sẽ nằm trong tệp này và kết thúc bằng phần mở rộng thư viện C, chẳng
 hạn như .so

Gọi
$python setup.py install sẽ
cài đặt mô-đun trong tệp gói trang web của bạn.

Xem phần distutils của
'Mở rộng và những trình thông dịch Python' tại docs.python.org để biết thêm
thông tin.
'
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```
từ thiết lập nhập distutils.core , Tiện ích mở rộng

module1 = Tiện ích mở rộng('spam', nguồn=['spammodule.c'], include_dirs=['/usr/local/lib'])

setup(name = 'spam',
      version='1.0',
      description='Đây là gói thư rác của tôi',
      ext_modules = [module1])
```

Sau khi mô-đun thư rác được nhập vào python, bạn có thể gọi logit qua spam.logit. Lưu ý rằng hàm được sử dụng ở trên không thể được áp dụng nguyên trạng cho các mảng có nhiều mảng. Để làm như vậy chúng ta phải gọi numpy.vectorize trên đó. Ví dụ: nếu trình thông dịch python được mở trong tệp chứa thư viện thư rác hoặc thư rác đã được cài đặt, người ta có thể thực hiện các lệnh sau:

```
>>> nhập numpy dưới dạng
np >>> nhập thư
rác >>> spam.logit(0)
-inf
>>> thông tin
spam.logit(1)
>>> spam.logit(0.5) 0.0

>>> x = np.linspace(0,1,10) >>
spam.logit(x)
TypeError: chỉ các mảng có độ dài-1 mới có thể được chuyển đổi thành vô hướng Python >>> f =
np.vectorize(spam.logit) >>> f(x) array([
-

thông tin])


```

**CHỨC NĂNG LOGIT KẾT QUẢ KHÔNG NHANH CHÓNG!** numpy.vectorize chỉ đơn giản là lặp lại spam.logit. Vòng lặp được thực hiện ở cấp độ C, nhưng mảng numpy liên tục được phân tích cú pháp và xây dựng lại. Cái này đắt quá. Khi tác giả so sánh numpy.vectorize(spam.logit) với các ufuncs logit được xây dựng bên dưới, các ufuncs logit nhanh hơn gần như chính xác 4 lần. Tuy nhiên, tốc độ lớn hơn hoặc nhỏ hơn có thể thực hiện được tùy thuộc vào tính chất của chức năng.

### 7.3.3 Ví dụ NumPy ufunc cho một dtype

Để đơn giản, chúng tôi cung cấp ufunc cho một dtype duy nhất, gấp đôi 'f8'. Như trong phần trước, trước tiên chúng tôi cung cấp tệp .c và sau đó là tệp setup.py được sử dụng để tạo mô-đun chứa ufunc.

Vị trí trong mã tương ứng với các phép tính thực tế cho ufunc được đánh dấu bằng /\*BEGIN tính toán ufunc chính\*/ và /\*END tính toán ufunc chính\*. Mã ở giữa các dòng đó là thủ công phải được thay đổi để tạo ufunc của riêng bạn.

```
#include "Python.h"
#include "math.h"
#include "numpy/ndarraytypes.h"
#include "numpy/ufuncobject.h"
#include "numpy/npy_3kcompat.h"

'
* single_type_logit.c * Đây là mã
C để tạo mã của riêng bạn
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```

* NumPy ufunc cho hàm logit.
,
* Trong mã này, chúng tôi chỉ xác định ufunc cho * một dtype duy
nhất. Các phép tính phải * được thay thế để tạo ufunc cho * một hàm
khác được đánh dấu bằng BEGIN

* Và kết thúc.
,
* Bạn có thể tìm thấy thông tin chi tiết giải thích về API Python-C trong * 'Mở rộng
và nhúng' và 'API Python/C' tại * docs.python.org. ,

```

---

```

PyMethodDef tinh LogitMethods[] = {
    {NULL, KHÔNG, 0, KHÔNG}
}

/*
Định nghĩa vòng lặp phải đặt trước PyMODINIT_FUNC. ,

static void double_logit(char **args, npy_intp *dimensions,
                         bước npy_intp*, dữ liệu void* )

npy_intp tôi;
npy_intp n = kích thước [0]; char
*in = args[0], *out = args[1]; npy_intp
in_step = bước [0], out_step = bước [1];

đôi tmp;

vì (i = 0; i < n; i++) {
    /*BEGIN tính toán ufunc chính*/ tmp =
    *(double *)in; tmp /= 1-
    tmp; *((double
*)out) = log(tmp);
    /*KẾT THÚC tính toán ufunc chính*/

    trong += in_step;
    out += out_step;
}

/*
/*Đây là con trả tới hàm trên*/
PyUFuncGenericFunction funcs[1] = {&double_logit};

/* Đây là các kiểu đầu vào và trả về của logit.*/ static char type[2] =
{NPY_DOUBLE, NPY_DOUBLE};

khoảng trống tinh *dữ liệu [1] = {NULL};

#if PY_VERSION_HEX >= 0x03000000 cấu trúc tinh
PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "npufunc",
    Vô giá
    trị, -1,
    Phương thức Logit,
    VÔ GIÁ TRỊ,

```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```

vô GIÁ TRỊ,
vô GIÁ TRỊ,
vô GIÁ TRỊ

,

PyMODINIT_FUNC PyInit_npufunc(void) {

    PyObject *m, *logit, *d; m =
    PyModule_Create(&moduledesc); nếu (! m) {

        trả về NULL;
        ,

        import_array();
        import_umath();

        logit = PyUFunc_FromFuncAndData(funcs, dữ liệu, loại, 1, 1, 1,
                                         PyUFunc_None, "logit",
                                         "logit_docstring", 0);

        d = PyModule_GetDict(m);

        PyDict_SetItemString(d, "logit", logit);
        Py_DECREF(logit);

        trả lại m;
    } #khác
PyMODINIT_FUNC initnpufunc(void) {

    PyObject *m, *logit, *d;

    m = Py_InitModule("npufunc", LogitMethods); nếu (m == NULL) {

        trả lại;
        ,

        import_array();
        import_umath();

        logit = PyUFunc_FromFuncAndData(funcs, dữ liệu, loại, 1, 1, 1,
                                         PyUFunc_None, "logit",
                                         "logit_docstring", 0);

        d = PyModule_GetDict(m);

        PyDict_SetItemString(d, "logit", logit);
        Py_DECREF(logit);

    } #endif
}

```

Đây là tệp setup.py cho đoạn mã trên. Như trước đây, mô-đun có thể được xây dựng bằng cách gọi python setup.py build tại dấu nhắc lệnh hoặc cài đặt vào các gói trang web thông qua cài đặt python setup.py.

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

tập tin setup.py cho logit.c Lưu ý rằng  
 vì đây là một phần mở rộng khó hiểu nên chúng tôi sử dụng  
 numpy.distutils thay vì distutils từ thư viện chuẩn  
 python.

**Việc gọi**  
`$python setup.py build_ext --inplace` sẽ xây dựng thư viện tiện ích mở rộng trong tệp hiện tại.

**Việc gọi**  
`$python setup.py build` xây dựng một tệp trông giống như `./build/lib*`, trong đó `lib*` là tệp bắt đầu bằng `lib`. Thư viện sẽ nằm trong tệp này và kết thúc bằng phần mở rộng thư viện C, chẳng hạn như `.so`

**Gọi**  
`$python setup.py install` sẽ cài đặt mô-đun trong tệp gói trang web của bạn.

Xem phần distutils của 'Mở rộng và nhúng trình thông dịch Python' tại [docs.python.org](#) và tài liệu trên numpy.distutils để biết thêm thông tin.

```
cấu hình def (parent_package='', top_path=None): nhập numpy từ cấu hình nhập
numpy.distutils.misc_util

config = Cấu hình('npufunc_directory',
                    parent_package,
                    top_path)
config.add_extension('npufunc', ['single_type_logit.c'])

trả lại cấu hình

nếu __name__ == "__main__":
    từ thiết lập thiết lập nhập khẩu numpy.distutils.core (cấu
    hình = cấu hình)
```

Sau khi cài đặt xong phần trên, nó có thể được nhập và sử dụng như sau.

```
>>> nhập numpy dưới dạng
np >>> nhập npufunc
>>> npufunc.logit(0.5) 0.0

>>> a = np.linspace(0,1,5) >>>
npufunc.logit(a) array([-inf, -1.09861229, 0. , 1.09861229, thông tin])
```

### 7.3.4 Ví dụ NumPy ufunc với nhiều dtype

Cuối cùng, chúng tôi đưa ra một ví dụ về một ufunc đầy đủ, với các vòng lặp bên trong cho các chuỗi nửa float, float, double và long double. Như trong các phần trước, trước tiên chúng tôi cung cấp tệp .c và sau đó là tệp setup.py tương ứng.

Các vị trí trong mã tương ứng với các phép tính thực tế cho ufunc được đánh dấu bằng /\*BEGIN tính toán ufunc chính\*/ và /\*END tính toán ufunc chính\*/. Mã ở giữa các dòng đó là thủ chính phải được thay đổi để tạo ufunc của riêng bạn.

```
#include "Python.h"
#include "math.h"
#include "numpy/ndarraytypes.h"
#include "numpy/ufuncobject.h"
#include "numpy/halffloat.h"

/*
 * multi_type_logit.c * Đây là
mã C để tạo * NumPy ufunc của riêng bạn cho hàm logit.

,
* Mỗi hàm có dạng type_logit xác định hàm * logit cho một dtype có khối khác
nhau. Mỗi chức năng * trong số này phải được sửa đổi khi bạn * tạo ufunc của
riêng mình. Các phép tính phải * được thay thế để tạo ufunc cho * một hàm
khác được đánh dấu bằng BEGIN

* Và kết thúc.

,
* Bạn có thể tìm thấy thông tin chi tiết giải thích về API Python-C trong * 'Mở rộng
và nhúng' và 'API Python/C' tại * docs.python.org.

,
PyMethodDef tinh_LogitMethods[] = {
    {NULL, KHONG, 0, KHONG}
,

/* Định nghĩa vòng lặp phải đặt trước PyMODINIT_FUNC. ,

static void long_double_logit(char **args, npy_intp *dimensions,
                                bước npy_intp*, dữ liệu void* )
,
npy_intp tôi;
npy_intp n = kích thước [0]; char
*in = args[0], *out=args[1]; npy_intp
in_step = bước [0], out_step = bước [1];

tmp dài gấp đôi ;

vì (i = 0; i < n; i++) {
    /*BEGIN tính toán ufunc chính*/ tmp = *(long
double *)in; tmp /= 1-tmp; *((dài
gấp đôi *)out) =
    logl(tmp);
/*KẾT THÚC tính toán ufunc chính*/
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```

trong += in_step;
out += out_step;

'

static void double_logit(char **args, npy_intp *dimensions,
                         bước npy_intp* , dữ liệu void* )
{
    npy_intp tói;
    npy_intp n = kích thước [0]; char
    *in = args[0], *out = args[1]; npy_intp
    in_step = bước [0], out_step = bước [1];

    đôi tmp;

    vì (i = 0; i < n; i++) {
        /*BEGIN tính toán ufunc chính*/ tmp =
        *(double *)in; tmp /= 1-
        tmp; *((double
        *)out) = log(tmp);
        /*KẾT THÚC tính toán ufunc chính*/

        trong += in_step;
        out += out_step;
    }

    static void float_logit(char **args, npy_intp *dimensions,
                           bước npy_intp* , dữ liệu void* )
    {
        npy_intp tói;
        npy_intp n = kích thước [0]; char
        *in=args[0], *out = args[1]; npy_intp
        in_step = bước [0], out_step = bước [1];

        thả nổi tmp;

        vì (i = 0; i < n; i++) {
            /*BEGIN tính toán ufunc chính*/ tmp =
            *(float *)in; tmp /= 1-
            tmp; *((float
            *)out) = logf(tmp);
            /*KẾT THÚC tính toán ufunc chính*/

            trong += in_step;
            out += out_step;
        }

        static void Half_float_logit(char **args, npy_intp *dimensions,
                                     bước npy_intp* , dữ liệu void* )
        {
            npy_intp tói;
            npy_intp n = kích thước [0]; char
            *in = args[0], *out = args[1]; npy_intp
            in_step = bước [0], out_step = bước [1];

```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```

thả nỗi tmp;

vì (i = 0; i < n; i++) {

    /*BEGIN tính toán ufunc chính*/ tmp =
    *(npy_half *)in; tmp =
    npy_half_to_float(tmp); tmp /= 1-
    tmp; tmp =
    logf(tmp);
    *((npy_half *)out) = npy_float_to_half(tmp);
    /*KẾT THÚC tính toán ufunc chính*/

    trong += in_step;
    out += out_step;
    ,

    /*Cái này đưa ra con trỏ tới các hàm trên*/
PyUFuncGenericFunction funcs[4] = {&half_float_logit,
                                    &float_logit,
                                    &double_logit,
                                    &long_double_logit};

các loại char tinh [8] = {NPY_HALF, NPY_HALF,
                         NPY_FLOAT,NPY_FLOAT,
                         NPY_DOUBLE,NPY_DOUBLE,
                         NPY_LONGLONG, NPY_LONGLONG}; khoảng trống tinh
*dữ liệu [4] = {NULL, NULL, NULL, NULL};

#if PY_VERSION_HEX >= 0x03000000 cấu trúc tinh
PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "npufunc",
    vô giá
    trí, -1,
    Phương thức Logit,
    vô GIÁ TRỊ,
    vô GIÁ TRỊ,
    vô GIÁ TRỊ,
    vô GIÁ TRỊ
    ,

    PyMODINIT_FUNC PyInit_npufunc(void) {

        PyObject *m, *logit, *d; m =
        PyModule_Create(&moduledef); if (!m) { trả về
        NULL;

        ,

        import_array();
        import_umath();

        logit = PyUFunc_FromFuncAndData(funcs, dữ liệu, loại, 4, 1, 1,
                                         PyUFunc_None, "đăng nhập",
                                         
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```

        "logit_docstring", 0);

d = PyModule_GetDict(m);

PyDict_SetItemString(d, "logit", logit);
Py_DECREF(logit);

trả lại m;

} #khác

PyMODINIT_FUNC initnpufunc(void) {

PyObject *m, *logit, *d;

m = Py_InitModule("npufunc", LogitMethods); nếu (m == NULL) {

    trả lại;

    import_array();
    import_umath();

    logit = PyUFunc_FromFuncAndData(funcs, dữ liệu, loại, 4, 1, 1,
                                    PyUFunc_None, "logit",
                                    "logit_docstring", 0);

    d = PyModule_GetDict(m);

    PyDict_SetItemString(d, "logit", logit);
    Py_DECREF(logit);
}

#endif

```

Đây là tệp setup.py cho đoạn mã trên. Như trước đây, môđun có thể được xây dựng bằng cách gọi python setup.py build tại dấu nháy lệnh hoặc cài đặt vào các gói trang web thông qua cài đặt python setup.py.

```

'
tập tin setup.py cho logit.c Lưu ý rằng
vì đây là một phần mở rộng khó hiểu nên chúng tôi sử dụng
numpy.distutils thay vì distutils từ thư viện chuẩn
python.

Việc gọi
$python setup.py build_ext --inplace sẽ xây
dụng thư viện tiện ích mở rộng trong tệp hiện tại.

Việc gọi
$python setup.py build sẽ xây
dụng một tệp trông giống như ./build/lib*, trong đó lib* là tệp bắt đầu bằng
lib. Thư viện sẽ nằm trong tệp này và kết thúc bằng phần mở rộng thư viện C,
chẳng hạn như .so

Gọi
$python setup.py install

```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

sẽ cài đặt mô-đun trong tệp gói trang web của bạn.

Xem phần distutils của  
'Mở rộng và nhúng trình thông dịch Python' tại docs.python.org và tài liệu  
trên numpy.distutils để biết thêm thông tin.

```

cấu hình def (parent_package='', top_path=None): nhập numpy từ
numpy.distutils.misc_util nhập cấu hình từ numpy.distutils.misc_util nhập
get_info

#Cần thiết cho loại d nữa nỗi. thông tin =
get_info('npymath')

config = Cấu hình('npufunc_directory',
parent_package,
top_path)
config.add_extension('npufunc',
['multi_type_logit.c'],
extra_info=info)

trả lại cấu hình

nếu __name__ == "__main__":
    từ thiết lập thiết lập nhập khẩu numpy.distutils.core (cấu
    hình = cấu hình)

```

Sau khi cài đặt xong phần trên, nó có thể được nhập và sử dụng như sau.

```

>>> nhập numpy dưới dạng
np >>> nhập npufunc
>>> npufunc.logit(0.5) 0.0

>>> a = np.linspace(0,1,5) >>>
npufunc.logit(a) array([-inf, -1.09861229, , 1.09861229, thông tin])

```

### 7.3.5 Ví dụ NumPy ufunc với nhiều đối số/giá trị trả về

Ví dụ cuối cùng của chúng tôi là một ufunc có nhiều đối số. Đây là một sửa đổi mã cho logit ufunc cho dữ liệu có một dtype duy nhất. Chúng tôi tính toán ( $A^*B$ ,  $\logit(A^*B)$ ).

Chúng tôi chỉ cung cấp mã C vì tệp setup.py hoàn toàn giống với tệp setup.py trong [Ví dụ NumPy ufunc cho một dtype](#), ngoại trừ dòng

```
config.add_extension('npufunc', ['single_type_logit.c'])
```

được thay thế bằng

```
config.add_extension('npufunc', ['multi_arg_logit.c'])
```

Tệp C được đưa ra dưới đây. Ufunc được tạo có hai đối số A và B. Nó trả về một bộ có phần tử đầu tiên là  $A^*B$  và phần tử thứ hai là  $\logit(A^*B)$ . Lưu ý rằng nó tự động hỗ trợ phát sóng cũng như tất cả các tính năng khác.

các thuộc tính của một hàm.

```
#include "Python.h"
#include "math.h"
#include "numpy/ndarraytypes.h"
#include "numpy/ufuncobject.h"
#include "numpy/halffloat.h"

/*
 * multi_arg_logit.c * Đây là
 * mã C để tạo * NumPy ufunc của riêng bạn cho nhiều đối số, nhiều *
 giá trị trả về ufunc. Những nơi thực hiện tính toán * ufunc được đánh dấu *
 kèm theo nhận xét.

'
* Bạn có thể tìm thấy thông tin chi tiết giải thích về API Python-C trong * 'Mở rộng
và nhúng' và 'API Python/C' tại * docs.python.org.

'
'

PyMethodDef tinh_LogitMethods[] = {
    {NULL, KHONG, 0, KHONG}
}

/*
Định nghĩa vòng lặp phải đặt trước PyMODINIT_FUNC. ,

static void double_logitprod(char **args, npy_intp *dimensions,
                             bước npy_intp* , dữ liệu void* )

'
npy_intp tôi;
npy_intp n = kích thước [0]; char
*in1 = args[0], *in2 = args[1]; char *out1 =
args[2], *out2 = args[3]; npy_intp in1_step =
bước [0], in2_step = bước [1]; npy_intp out1_step = bước [2],
out2_step = bước [3];

đôi tmp;

vì (i = 0; i < n; i++) {
    /*BEGIN tính toán ufunc chính*/ tmp =
    *(double *)in1; tmp *=
    *(double *)in2; *((double
*)out1) = tmp; *((double
*)out2) = log(tmp/(1-tmp));
    /*KẾT THÚC tính toán ufunc chính*/

    in1 += in1_step;
    in2 += in2_step;
    out1 += out1_step;
    out2 += out2_step;
}

'
/*Đây là con trỏ tới hàm trên*/
```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```

PyUFuncGenericFunction funcs[1] = {&double_logitprod};

/* Đây là các kiểu dữ liệu đầu vào và trả về của logit.*/

các loại char tinh [4] = {NPY_DOUBLE, NPY_DOUBLE,
                           NPY_DOUBLE, NPY_DOUBLE};

khoảng trống tinh *dữ liệu [1] = {NULL};

#if PY_VERSION_HEX >= 0x03000000 cấu trúc tinh
PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "npufunc",
    VÔ GIÁ TRỊ,
    -1,
    Phương thức Logit,
    VÔ GIÁ TRỊ,
    VÔ GIÁ TRỊ,
    VÔ GIÁ TRỊ,
    VÔ GIÁ TRỊ
},

```

```

PyMODINIT_FUNC PyInit_npufunc(void) {

    PyObject *m, *logit, *d; m =
    PyModule_Create(&moduledef); if (m) { trả về
    NULL;

    import_array();
    import_umath();

    logit = PyUFunc_FromFuncAndData(funcs, dữ liệu, loại, 1, 2, 2,
                                    PyUFunc_None, "logit",
                                    "logit_docstring", 0);

    d = PyModule_GetDict(m);

    PyDict_SetItemString(d, "logit", logit);
    Py_DECREF(logit);

    trả lại m;
} #khác
PyMODINIT_FUNC initnpufunc(void) {

    PyObject *m, *logit, *d;

    m = Py_InitModule("npufunc", LogitMethods); if (m == NULL) { trả
    về;

    import_array();
}

```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```

import_umath();

logit = PyUFunc_FromFuncAndData(funcs, dữ liệu, loại, 1, 2, 2,
                                 PyUFunc_None, "logit",
                                 "logit_docstring", 0);

d = PyModule_GetDict(m);

PyDict_SetItemString(d, "logit", logit);
Py_DECREF(logit);

} #endif

```

### 7.3.6 Ví dụ NumPy ufunc với các đối số dtype mảng có cấu trúc

Ví dụ này cho thấy cách tạo ufunc cho dtype mảng có cấu trúc. Trong ví dụ này, chúng tôi hiển thị một ufunc tầm thường để cộng hai mảng với dtype 'u8,u8,u8'. Quá trình này hơi khác so với các ví dụ khác vì lệnh gọi tới `PyUFunc_FromFuncAndData` không đăng ký đầy đủ ufuncs cho các kiểu dtype tùy chỉnh và các kiểu dtype mảng có cấu trúc. Chúng ta cũng cần gọi `PyUFunc_RegisterLoopForDescr` để hoàn tất việc thiết lập ufunc.

Chúng tôi chỉ cung cấp mã C vì tệp `setup.py` hoàn toàn giống với tệp `setup.py` trong [Ví dụ NumPy ufunc cho một dtype, ngoại trừ dòng](#)

```
config.add_extension('npufunc', ['single_type_logit.c'])
```

được thay thế bằng

```
config.add_extension('npufunc', ['add_triplet.c'])
```

Tệp C được đưa ra dưới đây.

```

#include "Python.h"
#include "math.h"
#include "numpy/ndarraytypes.h"
#include "numpy/ufuncobject.h"
#include "numpy/npy_3kcompat.h"

/*
 * add_triplet.c * Đây
 * là mã C để tạo * NumPy ufunc của riêng bạn cho kiểu dtype mảng có
 * cấu trúc.
 *
 * Bạn có thể tìm thấy thông tin chi tiết giải thích về API Python-C trong * 'Mở rộng
 * và nhúng' và 'API Python/C' tại * docs.python.org. ,
 */

PyMethodDef tinh StructUfuncTestMethods[] = {
    {NULL, KHÔNG, 0, KHÔNG}
}

/*
 * Định nghĩa vòng lặp phải đặt trước PyMODINIT_FUNC. ,

static void add_uint64_triplet(char **args, npy_intp *dimensions,

```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```

        bước npy_intp* , dữ liệu void* )

'
npy_intp tői;
npy_intp is1=bước[0];
npy_intp is2=bước[1];
npy_intp os=bước[2];
npy_intp n=kích thước[0];
uint64_t *x, *y, *z;

char *i1=args[0];
char *i2=args[1];
char *op=args[2];

vì (i = 0; i < n; i++) {

    x = (uint64_t*)i1; y
    = (uint64_t*)i2; z =
    (uint64_t*)op;

    z[0] = x[0] + y[0];
    z[1] = x[1] + y[1];
    z[2] = x[2] + y[2];

    i1 += is1;
    i2 += is2;
    op += os;
'

'

/* Đây là con trỏ tới hàm trên */
PyUFuncGenericFunction funcs[1] = {&add_uint64_triplet};

/* Đây là các kiểu dữ liệu đầu vào và trả về của add_uint64_triplet. */ các loại char tinh [3] =
{NPY_UINT64, NPY_UINT64, NPY_UINT64};

khoảng trống tinh *dữ liệu [1] = {NULL};

#if được xác định (NPY_PY3K)
cấu trúc tinh PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "struct_ufunc_test",
    vô giá
    trị, -1,
    Phương thức StructUfuncTest,
    vô GIÁ TRỊ,
    vô GIÁ TRỊ,
    vô GIÁ TRỊ,
    vô GIÁ TRỊ

',
#endif

#if được xác định(NPY_PY3K)
PyMODINIT_FUNC PyInit_struct_ufunc_test(void) #else

PyMODINIT_FUNC initstruct_ufunc_test(void) #endif

'

```

(tiếp tục trên trang tiếp theo)

(tiếp theo trang trước)

```

PyObject *m, *add_triplet, *d;
PyObject *dtype_dict;
PyArray_Descr *dtype;
PyArray_Descr *dtypes[3];

#if được xác định(NPY_PY3K)
    m = PyModule_Create(&moduledesc); #khác

    m = Py_InitModule("struct_ufunc_test", StructUfuncTestMethods); #endif

    if (m == NULL) { #if
        được xác định(NPY_PY3K)
            trả về NULL;
    #khác
            trả lại;
    #endif
    }

    import_array();
    import_umath();

/* Tạo một đối tượng ufunc mới */
add_triplet
= PyUFunc_FromFuncAndData(NULL, NULL, NULL, 0, 2, 1,
                           PyUFunc_None, "add_triplet",
                           "add_triplet_docstring", 0);

dtype_dict = Py_BuildValue("[({s, s}, {s, s}, {s, s}]", "f0", "u8",
                           "f1", "u8", "f2", "u8");
PyArray_DescrConverter(dtype_dict, &dtype);
Py_DECREF(dtype_dict);

dtypes[0] = dtype;
dtypes[1] = dtype;
dtypes[2] = dtype;

/* Đăng ký ufunc cho dtype có cấu trúc */
PyUFunc_RegisterLoopForDescr(add_triplet, dtype,
                               &add_uint64_triplet,
                               dtypes,
                               VÔ GIÁ TRỊ);

d = PyModule_GetDict(m);

PyDict_SetItemString(d, "add_triplet", add_triplet);
Py_DECREF(add_triplet); #if được
xác định(NPY_PY3K) trả về m;
#endif

```

Đối tượng ufunc được trả về là đối tượng Python có thể gọi được. Nó phải được đặt trong một từ điển (mô-đun) có cùng tên như được sử dụng trong đối số tên của quy trình tạo ufunc. Ví dụ sau được điều chỉnh từ mô-đun umath

```

PyUFuncGenericFunction tinh atan2_functions[] = {
    PyUFunc_ff_f, PyUFunc_dd_d,
    PyUFunc_gg_g, PyUFunc_00_0_method}; khoảng trống tinh*
atan2_data[] = {
    (void *)atan2f,(void *)atan2, (void
    *)atan2l,(void *)"arctan2"};
char tinh atan2_signatures[] = {
    NPY_FLOAT,NPY_FLOAT,NPY_FLOAT,
    NPY_DOUBLE, NPY_DOUBLE, NPY_DOUBLE,
    NPY_LONG NHÂN ĐÔI, NPY_LONG NHÂN ĐÔI, NPY_LONG NHÂN ĐÔI
    NPY_OBJECT DỰ ÁN, NPY_OBJECT DỰ ÁN, NPY_OBJECT};

/* trong mã khởi tạo mô-đun */
PyObject *f, *dict, *module;
.

dict = PyModule_GetDict(module);
.

f = PyUFunc_FromFuncAndData(atan2_functions, atan2_data,
    atan2_signatures, 4, 2, 1,
    PyUFunc_None, "arctan2",
    "arctan(x1/x2) an toàn và chính xác", 0);
PyDict_SetItemString(dict, "arctan2", f);
Py_DECREF(f);
.

```

## 7.4 Ngoài những điều cơ bản

Hành trình khám phá không phải là tìm kiếm những cảnh quan mới mà là có được  
đôi mắt mới.  
—Marcel Proust

Khám phá là nhìn thấy những gì người khác đã thấy và nghĩ những điều chưa  
ai từng nghĩ.  
—Albert Szent-Gyorgyi

### 7.4.1 Lặp lại các phần tử trong mảng

#### Lặp lại cơ bản

Một yêu cầu thuật toán phổ biến là có thể duyệt qua tất cả các phần tử trong một mảng đa chiều. Đối tượng trình vòng lặp mảng làm cho việc này trở nên dễ dàng thực hiện theo cách chung, phù hợp với các mảng có kích thước bất kỳ. Đường nhiên, nếu bạn biết số thứ nguyên mình sẽ sử dụng thì bạn luôn có thể viết các vòng lặp for lồng nhau để hoàn thành việc lặp. Tuy nhiên, nếu bạn muốn viết mã hoạt động với bất kỳ số thứ nguyên nào thì bạn có thể sử dụng trình vòng lặp mảng. Một đối tượng lặp mảng được trả về khi truy cập thuộc tính .flat của một mảng.

Cách sử dụng cơ bản là gọi `PyArray_IterNew ( array )` trong đó mảng là một đối tượng `ndarray` (hoặc một trong các lớp con của nó). Đối tượng được trả về là một đối tượng lặp mảng (cùng một đối tượng được trả về bởi thuộc tính `.flat` của `ndarray`). Đối tượng này thường được truyền tới `PyArrayIterObject*` để có thể truy cập các thành viên của nó. Các thành viên duy nhất cần thiết là `iter->size` chứa tổng kích thước của mảng, `iter->index`, chứa chỉ mục 1-d hiện tại trong mảng và `iter->dataptr` là con trỏ tới dữ liệu cho phần tử hiện tại của mảng. Đôi khi, việc truy cập `iter->ao` là một con trỏ tới đối tượng `ndarray` bên dưới cũng rất hữu ích.

Sau khi xử lý dữ liệu ở phần tử hiện tại của mảng, phần tử tiếp theo của mảng có thể được lấy bằng macro `PyArray_ITER_NEXT` (`iter`). Việc lặp lại luôn tiến hành theo kiểu liên kè kiểu C (chỉ số cuối cùng thay đổi nhanh nhất). `PyArray_ITER_GOTO` (`iter, Destination`) có thể được sử dụng để chuyển đến một điểm cụ thể trong mảng, trong đó đích là một mảng kiểu dữ liệu `npy_intp` có khoảng trống để xử lý ít nhất số lượng kích thước trong mảng bên dưới. Đôi khi, rất hữu ích khi sử dụng `PyArray_GOTO1D` (`iter, index`) sẽ chuyển sang chỉ mục 1-d được cung cấp bởi giá trị của chỉ mục. Tuy nhiên, cách sử dụng phổ biến nhất được đưa ra trong ví dụ sau.

```
PyObject *obj; /* giả định là một đối tượng ndarray nào đó */
PyArrayIterObject *iter;
'
iter = (PyArrayIterObject *)PyArray_IterNew(obj); if (iter == NULL)
    bị lỗi; /* Giả sử lỗi có mã dọn dẹp */ while (iter->index < iter->size) {

    /* làm gì đó với dữ liệu tại đó->dataptr */
    PyArray_ITER_NEXT(nó);
'
'
```

Bạn cũng có thể sử dụng `PyArrayIter_Check` (`obj`) để đảm bảo bạn có một đối tượng iterator và `PyArray_ITER_RESET` (`iter`) để đặt lại một đối tượng iterator về đầu mảng.

Cần nhấn mạnh vào thời điểm này rằng bạn có thể không cần trình lặp mảng nếu mảng của bạn đã liền kề (sử dụng trình lặp mảng sẽ hoạt động nhưng sẽ chậm hơn mã nhanh nhất bạn có thể viết). Mục đích chính của các trình lặp mảng là đóng gói phép lặp trên các mảng N chiều với các bước tùy ý. Chúng được sử dụng ở rất nhiều nơi trong chính mã nguồn NumPy. Nếu bạn đã biết mảng của mình là liền kề (Fortran hoặc C), thì chỉ cần thêm kích thước phần tử vào biến con trả đang chạy sẽ giúp bạn duyệt mảng rất hiệu quả. Nói cách khác, mã như thế này có thể sẽ nhanh hơn đối với bạn trong trường hợp liền kề (giả sử là gấp đôi).

```
kích thước npy_intp;
gấp đôi *dptr; /* có thể tạo bất kỳ loại biến nào */ size = PyArray_SIZE(obj); dptr
= PyArray_DATA(obj); while(size--) { /
* làm gì đó với dữ liệu tại dptr */
dptr++;

'
'
```

#### Lặp lại trên tất cả trừ một trục

Một thuật toán phổ biến là lặp qua tất cả các phần tử của một mảng và thực hiện một số chức năng với từng phần tử bằng cách thực hiện lệnh gọi hàm. Vì lệnh gọi hàm có thể tồn thời gian, nên một cách để tăng tốc loại thuật toán này là viết hàm sao cho nó lấy một vectơ dữ liệu rồi viết phép lặp để lệnh gọi hàm được thực hiện cho toàn bộ chiều dữ liệu tại một thời điểm. Điều này làm tăng lượng công việc được thực hiện trên mỗi lệnh gọi hàm, do đó giảm chi phí hoạt động của lệnh gọi hàm xuống một phần nhỏ (`er`) trong tổng thời gian. Ngay cả khi phần bên trong của vòng lặp được thực hiện mà không có lệnh gọi hàm thì vẫn có thể thuận lợi khi thực hiện vòng lặp bên trong theo kích thước có số phần tử cao nhất để tận dụng các cải tiến tốc độ có sẵn trên các bộ vi xử lý sử dụng đường ống để nâng cao các hoạt động cơ bản.

`PyArray_IterAllButAxis` (`array, &dim`) xây dựng một đối tượng iterator được sửa đổi để nó không lặp lại theo kích thước được chỉ định bởi `dim`. Hạn chế duy nhất đối với đối tượng iterator này là không thể sử dụng macro `PyArray_Iter_GOTO1D` (`it, ind`) (do đó, lặp chỉ mục phẳng cũng sẽ không hoạt động nếu bạn chuyển đổi đối tượng này trở lại Python - vì vậy bạn không nên làm điều này). Lưu ý rằng đối tượng được trả về từ quy trình này vẫn thường được chuyển thành `PyArrayIterObject` \*. Tất cả những gì đã được thực hiện là sửa đổi các bước và kích thước của trình vòng lặp được trả về để mô phỏng việc lặp qua mảng `[. . . , 0, . . . , ]` trong đó `0` được đặt trên chiều mờ. Nếu độ mờ là âm thì kích thước có trực lớn nhất sẽ được tìm thấy và sử dụng.

### Lặp lại trên nhiều mảng

Rất thường xuyên, người ta mong muốn lặp lại nhiều mảng cùng một lúc. Các chức năng phổ quát là một ví dụ về loại hành vi này. Nếu tất cả những gì bạn muốn làm là lặp qua các mảng có hình dạng giống nhau thì chỉ cần tạo một số đối tượng iterator là quy trình chuẩn. Ví dụ: đoạn mã sau lặp qua hai mảng được giả sử là có cùng hình dạng và kích thước (thực tế obj1 chỉ cần có tổng số phần tử ít nhất bằng tổng số phần tử như obj2):

```
/* Người ta đã giả định rằng obj1 và obj2 là các mảng có cùng
   hình dạng và kích thước.

iter1 = (PyArrayIterObject *)PyArray_IterNew(obj1); if (iter1 == NULL)
bị lỗi; iter2 = (PyArrayIterObject
*)PyArray_IterNew(obj2); if (iter2 == NULL) bị lỗi; /* giả sử iter1 bị
DECREF bị lỗi */ while (iter2->index < iter2->size) {

    /* xử lý với iter1->dataptr và iter2->dataptr */
    PyArray_ITER_NEXT(iter1);
    PyArray_ITER_NEXT(iter2);
}
```

### Phát sóng trên nhiều mảng

Khi có nhiều mảng tham gia vào một phép toán, bạn có thể muốn sử dụng cùng các quy tắc phát sóng mà các phép toán (tức là ufuncs) sử dụng. Điều này có thể được thực hiện dễ dàng bằng cách sử dụng PyArrayMultiIterObject. Đây là đối tượng được trả về từ lệnh Python numpy.broadcast và nó gần như dễ sử dụng từ C. Hàm PyArray\_MultiIterNew ( n, ... ) được sử dụng (với n đối tượng đầu vào thay cho ... ). Các đối tượng đầu vào có thể là mảng hoặc bất cứ thứ gì có thể chuyển đổi thành mảng. Một con trỏ tới PyArrayMultiIterObject được trả về. Việc truyền rộng đã được thực hiện để điều chỉnh các vòng lặp sao cho tất cả những gì cần thực hiện để chuyển sang phần tử tiếp theo trong mỗi mảng là để gọi PyArray\_ITER\_NEXT cho mỗi đầu vào. Việc tăng dần này được thực hiện tự động bởi macro PyArray\_MultiIter\_NEXT ( obj ) (có thể xử lý một obj đa năng dưới dạng PyArrayMultiObject \* hoặc PyObject , Dữ liệu từ số đầu vào i có sẵn bằng cách sử dụng PyArray\_MultiIter\_DATA ( obj, i ) và tổng kích thước (được phát sóng) là PyArray\_MultiIter\_SIZE ( obj ). Sau đây là một ví dụ về việc sử dụng tính năng này.

```
mobj = PyArray_MultiIterNew(2, obj1, obj2); kích thước
= PyArray_MultiIter_SIZE(obj); while(size--)
{ ptr1 =
    PyArray_MultiIter_DATA(mobj, 0); ptr2 =
    PyArray_MultiIter_DATA(mobj, 1); /* mã sử dụng
nội dung của ptr1 và ptr2 */
    PyArray_MultiIter_NEXT(mobj));
}
```

Hàm PyArray\_RemoveSmallest ( multi ) có thể được sử dụng để lấy một đối tượng có nhiều vòng lặp và điều chỉnh tất cả các vòng lặp để việc lặp không diễn ra trên chiều lớn nhất (nó làm cho chiều đó có kích thước 1). Mã được lặp lại sử dụng các con trỏ rất có thể cũng sẽ cần dữ liệu về bước tiến cho mỗi trình vòng lặp.

Thông tin này được lưu trữ trong multi->iters[i]->sai bước.

Có một số ví dụ về việc sử dụng multi-iterator trong mã nguồn NumPy vì nó làm cho việc viết mã phát sóng N chiều trở nên rất đơn giản. Duyệt nguồn để biết thêm ví dụ.

#### 7.4.2 Kiểu dữ liệu do người dùng xác định

NumPy đi kèm với 24 kiểu dữ liệu dựng sẵn. Mặc dù điều này bao gồm phần lớn các trường hợp sử dụng có thể xảy ra, nhưng có thể hình dung rằng người dùng có thể có nhu cầu về loại dữ liệu bổ sung. Có một số hỗ trợ để thêm kiểu dữ liệu bổ sung vào hệ thống NumPy. Kiểu dữ liệu bổ sung này sẽ hoạt động giống như kiểu dữ liệu thông thường ngoại trừ ufuncs phải có vòng lặp 1-d được đăng ký để xử lý riêng biệt. Ngoài ra, việc kiểm tra xem các loại dữ liệu khác có thể được truyền “an toàn” đến và từ loại mới này hay không sẽ luôn trả về “có thể truyền” trừ khi bạn cũng đăng ký loại dữ liệu mới nào có thể được truyền tới và từ đó.

Mã nguồn NumPy bao gồm một ví dụ về kiểu dữ liệu tùy chỉnh như một phần của bộ thử nghiệm. Tệp `_rational_tests.c.src` trong thư mục mã nguồn `numpy/core/src/umath/` chứa bản triển khai của loại dữ liệu biểu thị số hữu tỷ là tỷ lệ của hai số nguyên 32 bit.

##### Thêm kiểu dữ liệu mới

Để bắt đầu sử dụng kiểu dữ liệu mới, trước tiên bạn cần xác định một kiểu Python mới để giữ các giá trị vô hướng của kiểu dữ liệu mới của bạn. Có thể chấp nhận kế thừa từ một trong các mảng vô hướng nếu kiểu mới của bạn có bố cục tương thích nhị phân. Điều này sẽ cho phép kiểu dữ liệu mới của bạn có các phương thức và thuộc tính của mảng vô hướng. Các kiểu dữ liệu mới phải có kích thước bộ nhớ cố định (nếu bạn muốn xác định kiểu dữ liệu cần biểu diễn linh hoạt, như số có độ chính xác thay đổi, thì hãy sử dụng con trỏ tới đối tượng làm kiểu dữ liệu). Bố cục bộ nhớ của cấu trúc đối tượng cho kiểu Python mới phải là `PyObject_HEAD`, theo sau là bộ nhớ có kích thước cố định cần thiết cho kiểu dữ liệu. Ví dụ: cấu trúc phù hợp cho loại Python mới là:

```
cấu trúc typedef {
    PyObject_HEAD;
    some_data_type hình trái ngược; /
    *tên có thể là bất cứ thứ gì bạn muốn */
} PySomeDataTypeObject;
```

Sau khi bạn đã xác định một đối tượng kiểu Python mới, bạn phải xác định cấu trúc `PyArray_Descr` mới mà thành viên kiểu đối tượng sẽ chứa một con trỏ tới kiểu dữ liệu bạn vừa xác định. Ngoài ra, các hàm bắt buộc trong thành viên `.f` phải được xác định: `nonzero`, `Copyswap`, `Copyswapn`, `setitem`, `getitem` và `cast`. Tuy nhiên, bạn xác định càng nhiều hàm trong thành viên `.f` thì kiểu dữ liệu mới sẽ càng hữu ích. Điều rất quan trọng là khởi tạo các hàm không sử dụng thành `NULL`. Điều này có thể đạt được bằng cách sử dụng `PyArray_InitArrFuncs (f)`.

Khi cấu trúc `PyArray_Descr` mới được tạo và chứa đầy thông tin cần thiết cũng như các hàm hữu ích, bạn gọi `PyArray_RegisterDataType (new_descr)`. Giá trị trả về từ lệnh gọi này là một số nguyên cung cấp cho bạn một `type_number` duy nhất chỉ định loại dữ liệu của bạn. Số loại này phải được mô-đun của bạn lưu trữ và cung cấp để các mô-đun khác có thể sử dụng nó để nhận dạng loại dữ liệu của bạn (cơ chế khác để tìm số loại dữ liệu do người dùng xác định là tìm kiếm dựa trên tên của loại- đối tượng được liên kết với kiểu dữ liệu bằng cách sử dụng `PyArray_TypeNumFromName` ).

##### Đăng ký chức năng truyền

Bạn có thể muốn cho phép các kiểu dữ liệu dựng sẵn (và các kiểu dữ liệu khác do người dùng xác định) được tự động chuyển sang kiểu dữ liệu của bạn. Để thực hiện được điều này, bạn phải đăng ký chức năng truyền với kiểu dữ liệu mà bạn muốn có thể truyền từ đó. Điều này yêu cầu viết các hàm truyền cấp độ thấp cho mỗi chuyển đổi mà bạn muốn hỗ trợ và sau đó đăng ký các hàm này với bộ mô tả kiểu dữ liệu. Chức năng truyền cấp thấp có chữ ký.

```
void castfunc(void* from, void* to, npy_intp n, void* fromarr, void* toarr)
Truyền n phần tử từ loại này sang loại khác. Dữ liệu cần truyền nằm trong một đoạn bộ nhớ liền kề, được hoán đổi chính xác và căn chỉnh được trả đến bởi từ đó. Bộ đệm để truyền tới cũng liền kề, được hoán đổi và căn chỉnh chính xác. Các đối số fromarr và toarr chỉ nên được sử dụng cho các mảng có kích thước phần tử linh hoạt (chuỗi, unicode, void).
```

Một ví dụ castfunc là:

```
khoảng trống tinh
double_to_float(double *from, float* to, npy_intp n, void* ign1,
                void* ign2) {
    trong khi (n--) {
        (*to++) = (gấp đôi) *(từ++);
    }
}
```

Điều này sau đó có thể được đăng ký để chuyển đổi gấp đôi thành số float bằng mã:

```
nhân đôi = PyArray_DescrFromType(NPY_DOUBLE);
PyArray_RegisterCastFunc(doub, NPY_FLOAT,
    (PyArray_VectorUnaryFunc *)double_to_float);
Py_DECREF(gấp đôi);
```

#### Đăng ký quy tắc cưỡng chế

Theo mặc định, tất cả các kiểu dữ liệu do người dùng xác định không được coi là có thể chuyển một cách an toàn sang bất kỳ kiểu dữ liệu dựng sẵn nào. Ngoài ra, các kiểu dữ liệu dựng sẵn không được coi là có thể truyền an toàn sang các kiểu dữ liệu do người dùng xác định. Tình huống này giới hạn khả năng các kiểu dữ liệu do người dùng xác định tham gia vào hệ thống cưỡng chế được ufuncs sử dụng và các thời điểm khác khi việc ép buộc tự động diễn ra trong NumPy. Điều này có thể được thay đổi bằng cách đăng ký các kiểu dữ liệu có thể truyền an toàn từ một đối tượng kiểu dữ liệu cụ thể. Nên sử dụng hàm `PyArray_RegisterCanCast` (`from_descr`, `totype_number`, `scalarkind`) để chỉ định rằng đối tượng kiểu dữ liệu `from_descr` có thể được chuyển sang kiểu dữ liệu có số kiểu `totype_number`. Nếu bạn không cố gắng thay đổi các quy tắc cưỡng chế vô hướng thì hãy sử dụng `NPY_NOSCALAR` cho đối số vô hướng.

Nếu bạn muốn cho phép kiểu dữ liệu mới của mình cũng có thể chia sẻ các quy tắc cưỡng chế vô hướng, thì bạn cần chỉ định hàm vô hướng trong thành viên `".f"` của đối tượng kiểu dữ liệu để trả về loại vô hướng của dữ liệu mới. -type nên được xem là (giá trị của vô hướng có sẵn cho hàm đó). Sau đó, bạn có thể đăng ký các kiểu dữ liệu có thể được chuyển thành riêng biệt cho từng loại vô hướng có thể được trả về từ kiểu dữ liệu do người dùng xác định của bạn. Nếu bạn không đăng ký xử lý cưỡng chế vô hướng thì tất cả các kiểu dữ liệu do người dùng xác định sẽ được xem là `NPY_NOSCALAR`.

#### Đăng ký vòng lặp ufunc

Bạn cũng có thể muốn đăng ký các vòng lặp ufunc cấp thấp cho kiểu dữ liệu của mình để một mảng trong kiểu dữ liệu của bạn có thể áp dụng toán học một cách liền mạch. Đăng ký một vòng lặp mới có cùng chữ ký `arg_types`, âm thầm thay thế mọi vòng lặp đã đăng ký trước đó cho kiểu dữ liệu đó.

Trước khi bạn có thể đăng ký vòng lặp 1-d cho ufunc, ufunc phải được tạo trước đó. Sau đó, bạn gọi `PyUFunc_RegisterLoopForType` ( . . . ) với thông tin cần thiết cho vòng lặp. Giá trị trả về của hàm này là 0 nếu quá trình thành công và -1 với điều kiện lỗi được đặt nếu quá trình không thành công.

### 7.4.3 Phân nhóm ndarray trong C

Một trong những tính năng ít được sử dụng hơn đã xuất hiện trong Python kể từ phiên bản 2.2 là khả năng phân lớp con trong C. Cơ sở này là một trong những lý do quan trọng để dựa trên NumPy dựa trên cơ sở mã C đã có trong C. Loại phụ trong C cho phép linh hoạt hơn nhiều trong việc quản lý bộ nhớ. Việc gõ phụ trong C không khó ngay cả khi bạn chỉ có hiểu biết sơ bộ về cách tạo kiểu mới cho Python. Mặc dù việc gõ phụ từ một kiểu cha mẹ là dễ dàng nhất, nhưng cũng có thể gõ phụ từ nhiều kiểu cha mẹ. Đa kế thừa trong C thường ít hữu ích hơn so với trong Python vì hạn chế đối với các kiểu con Python là chúng có bố cục bộ nhớ tương thích nhị phân. Có lẽ vì lý do này, việc phân loại phụ dễ dàng hơn một chút so với loại cha mẹ đơn lẻ.

Tất cả các cấu trúc C tương ứng với các đối tượng Python phải bắt đầu bằng `PyObject_HEAD` (hoặc `PyObject_VAR_HEAD`). Theo cách tương tự, bất kỳ loại phụ nào cũng phải có cấu trúc C bắt đầu với bộ cục bộ nhớ giống hệt như loại cha mẹ (hoặc tất cả các loại cha mẹ trong trường hợp đa kế thừa). Lý do cho điều này là Python có thể có gắng truy cập một thành viên của cấu trúc kiểu con như nó có cấu trúc cha (tức là nó sẽ chuyển một con trỏ đã cho tới một con trỏ tới cấu trúc cha và sau đó hủy đăng ký một trong các thành viên của nó). Nếu bộ cục bộ nhớ không tương thích thì nỗ lực này sẽ gây ra hành vi không thể đoán trước (cuối cùng dẫn đến vi phạm bộ nhớ và hỏng chương trình).

Một trong những phần tử trong `PyObject_HEAD` là một con trỏ tới cấu trúc kiểu đối tượng. Một kiểu Python mới được tạo bằng cách tạo một cấu trúc đối tượng kiểu mới và điền vào đó các hàm và con trỏ để mô tả hành vi mong muốn của kiểu đó. Thông thường, cấu trúc C mới cũng được tạo để chứa thông tin cụ thể về phiên bản cần thiết cho từng đối tượng thuộc loại. Ví dụ: `&PyArray_Type` là một con trỏ tới bảng kiểu đối tượng cho `ndarray` trong khi biến `PyArrayObject *` là một con trỏ tới một thể hiện cụ thể của `ndarray` (một trong các thành viên của cấu trúc `ndarray` lần lượt là một con trỏ tới kiểu - bảng đối tượng `&PyArray_Type`). Cuối cùng `PyType_Ready (<pointer_to_type_object>)` phải được gọi cho mọi loại Python mới.

#### Tạo các loại phụ

Để tạo một kiểu con, phải tuân theo một quy trình tương tự, ngoại trừ chỉ những hành vi khác nhau mới yêu cầu các mục mới trong cấu trúc đối tượng kiểu. Tất cả các mục khác có thể là NULL và sẽ được `PyType_Ready` điền vào với các chức năng thích hợp từ (các) loại cha mẹ. Cụ thể, để tạo một kiểu con trong C, hãy làm theo các bước sau:

1. Nếu cần, hãy tạo cấu trúc C mới để xử lý từng phiên bản thuộc loại của bạn. Cấu trúc C điển hình sẽ là:

```
typedef _new_struct {
    Cơ sở PyArrayObject; /*có
    điều mới ở đây*/
} NewArrayObject;
```

Lưu ý rằng `PyArrayObject` đầy đủ được sử dụng làm mục nhập đầu tiên để đảm bảo rằng bộ cục bộ phân của các phiên bản thuộc loại mới giống hệt với `PyArrayObject`.

2. Điền vào cấu trúc đối tượng kiểu Python mới với các con trỏ tới các hàm mới sẽ ghi đè hành vi mặc định trong khi vẫn giữ nguyên bất kỳ hàm nào không được thực hiện (hoặc NULL). Phần tử `tp_name` phải khác.
3. Điền vào thành viên `tp_base` của cấu trúc đối tượng kiểu mới bằng một con trỏ tới đối tượng kiểu cha (chính). Đối với đa kế thừa, cũng điền vào thành viên `tp_bases` một bộ chứa tất cả các đối tượng cha theo thứ tự chúng sẽ được sử dụng để xác định tính kế thừa. Hãy nhớ rằng, tất cả các kiểu cha mẹ phải có cùng cấu trúc C để tính năng đa kế thừa hoạt động bình thường.
4. Gọi `PyType_Ready (<con trỏ_to_new_type>)`. Nếu hàm này trả về số âm thì đã xảy ra lỗi và kiểu dữ liệu không được khởi tạo. Nếu không, loại này đã sẵn sàng để sử dụng. Nói chung, điều quan trọng là phải đặt tham chiếu đến loại mới vào từ điển mô-đun để có thể truy cập nó từ Python.

Bạn có thể tìm hiểu thêm thông tin về cách tạo loại phụ trong C bằng cách đọc PEP 253 (có sẵn tại <https://www.python.org/dev/peps/pep-0253> ).

### Các tính năng cụ thể của kiểu gõ phụ ndarray

Một số phương thức và thuộc tính đặc biệt được mang sử dụng để tạo điều kiện thuận lợi cho việc tương tác giữa các kiểu con với kiểu ndarray cơ sở.

#### Phương thức `__array_finalize__`

##### `ndarray.__array_finalize__`

Một số hàm tạo mảng của ndarray cho phép tạo đặc tả của một kiểu con cụ thể. Điều này cho phép các loại phụ được xử lý liền mạch trong nhiều quy trình. Tuy nhiên, khi một loại phụ được tạo theo kiểu như vậy, cả phương thức `__new__` lẫn phương thức `__init__` đều không được gọi. Thay vào đó, loại phụ được phân bổ và các thành viên cấu trúc cá thể thích hợp được diền vào. Cuối cùng, thuộc tính `__array_finalize__` được tra cứu trong từ điển đối tượng. Nếu nó hiện diện chứ không phải Không có thì nó có thể là CObject chứa con trỏ tới PyArray\_FinalizeFunc hoặc nó có thể là một phương thức lấy một đối số duy nhất (có thể là Không có).

Nếu thuộc tính `__array_finalize__` là CObject thì con trỏ phải là con trỏ tới hàm có chữ ký:

```
(int) (PyArrayObject *, PyObject *)
```

Đối số đầu tiên là loại phụ mới được tạo. Đối số thứ hai (nếu không phải là NULL) là mảng "cha" (nếu mảng được tạo bằng cách cắt hoặc một số thao tác khác trong đó có một cha mẹ có thể phân biệt rõ ràng).

Thói quen này có thể làm bất cứ điều gì nó muốn. Nó sẽ trả về -1 nếu có lỗi và 0 nếu không.

Nếu thuộc tính `__array_finalize__` không phải là None hay CObject, thì đó phải là một phương thức Python lấy mảng cha làm đối số (có thể là Không nếu không có cha) và không trả về gì. Các lỗi trong phương pháp này sẽ được bắt và xử lý.

#### Thuộc tính `__array_priority__`

##### `ndarray.__array_priority__`

Thuộc tính này cho phép xác định đơn giản nhưng linh hoạt loại phụ nào sẽ được coi là "chính" khi phát sinh một hoạt động liên quan đến hai hoặc nhiều loại phụ. Trong các hoạt động sử dụng các loại phụ khác nhau, loại phụ có thuộc tính `__array_priority__` lớn nhất sẽ xác định loại phụ của (các) đầu ra.

Nếu hai loại phụ có cùng `__array_priority__` thì loại phụ của đối số đầu tiên sẽ xác định đầu ra. Thuộc tính `__array_priority__` mặc định trả về giá trị 0,0 cho loại ndarray cơ sở và 1,0 cho loại phụ. Thuộc tính này cũng có thể được xác định bởi các đối tượng không phải là kiểu con của ndarray và có thể được sử dụng để xác định phương thức `__array_wrap__` nào sẽ được gọi cho đầu ra trả về.

#### Phương thức `__array_wrap__`

##### `ndarray.__array_wrap__` Bất kỳ lớp

hoặc kiểu nào cũng có thể định nghĩa phương thức này, phương thức này sẽ nhận một đối số ndarray và trả về một thể hiện của kiểu đó. Nó có thể được coi là đối lập với phương thức `__array__`. Phương thức này được ufuncs (và các hàm NumPy khác) sử dụng để cho phép các đối tượng khác đi qua. Đối với Python> 2.4, nó cũng có thể được sử dụng để viết một trình trang trí chuyển đổi một hàm chỉ hoạt động với ndarrays thành một hàm hoạt động với bất kỳ loại nào bằng các phương thức `__array__` và `__array_wrap__`.

## CHỈ SỐ MÔ-ĐUN PYTHON

N

numpy.doc.basics, 33  
numpy.doc.broadcasting, 53  
numpy.doc.byteswapping, 56  
numpy.doc.creation, 37  
numpy.doc.dispatch, 79  
numpy.doc.indexing, 47  
numpy.doc.misc, 97  
numpy.doc.structured\_arrays, 58  
numpy.doc.subclassing, 84  
numpy.lib.recfunctions, 68



## MỤC LỤC

**b**iểu tượng

`__array_finalize__` (thuộc tính ndarray), 162  
`__array_priority__` (thuộc tính ndarray), 162  
`__array_wrap__` (thuộc tính ndarray), 162

**M**ỘT

thêm mới  
`dtype`, 159, 160  
`ufunc`, 139, 142, 146, 155  
`append_fields()` (trong  
`numpy.lib.recfunctions`), 68  
`apply_along_fields()`  
`(TRONG`  
`numpy.lib.recfunctions`), 69  
trình lặp mảng, 156, 158  
`gán_fields_by_name()`  
`(TRONG`  
`numpy.lib.recfunctions`), 69

**B**

Tăng cường Python, 138  
phát sóng, 158

**C**

`castfunc` (hàm C), 159  
`ctypes`, 131, 137  
`cython`, 129, 131

**D**

`drop_fields()` (trong mô-đun `numpy.lib.recfunctions`),  
69  
`dtype`  
thêm mới, 159, 160

**E**

mô-đun mở rộng, 115, 122

**F**

`f2py`, 124, 128  
`find_duplicates()` (trong  
`numpy.lib.recfunctions`), 70  
(trong `Flatten_descr()`  
`numpy.lib.recfunctions`), 70

**G**

`get_fieldstructure()`  
`(TRONG`  
`numpy.lib.recfunctions`), 71  
`get_names()` (trong mô-đun `numpy.lib.recfunctions`), 71  
`get_names_flat()` (trong mô-đun  
`numpy.lib.recfunctions`), 71

**J**

`join_by()` (trong mô-đun `numpy.lib.recfunctions`), 72

**M**

`merge_arrays()` (trong mô-đun `numpy.lib.recfunctions`),  
73

**N**

`ndarray`  
phân nhóm, 160, 162  
`ndpointer()` (hàm tích hợp sẵn), 133  
`numpy.doc.basics` (mô-đun), 33  
`numpy.doc.broadcasting`(mô-đun), 53  
`numpy.doc.byteswapping` (mô-đun), 56  
`numpy.doc.creation`(mô-đun), 37  
`numpy.doc.dispatch`(mô-đun), 79  
`numpy.doc.indexing` (mô-đun), 47  
`numpy.doc.misc` (mô-đun), 97  
`numpy.doc.structured_arrays` (mô-đun), 58  
`numpy.doc.subclassing`(mô-đun), 84  
`numpy.lib.recfunctions`(mô-đun), 68

**P**

`PyModule_AddIntConstant` (hàm C), 116  
`PyModule_AddObject` (hàm C), 116  
`PyModule_AddStringConstant` (hàm C), 116

**R**

`rec_append_fields()`  
`(trong numpy.lib.recfunctions)`, 73  
`rec_drop_fields()` (trong  
`numpy.lib.recfunctions`), 74  
`rec_join()` (trong mô-đun `numpy.lib.recfunctions`), 74  
`recursive_fill_fields()` (trong mô-đun  
`numpy.lib.recfunctions`), 74

đếm tham chiếu, 118, 119  
đổi tên fields() (trong numpy.lib.recfunctions), 75  
repack\_fields() (trong numpy.lib.recfunctions), 75  
require\_fields() (trong numpy.lib.recfunctions), 76

## S

NHẬM NHI, 138  
stack\_arrays() (trong mô-đun numpy.lib.recfunctions), 76  
có cấu trúc\_to\_unstructured() (trong mô-đun numpy.lib.recfunctions), 77  
phân nhóm ndarray, 160, 162  
uống rượu, 138  
  
bạn  
ufunc  
thêm mới, 139, 142, 146, 155  
unstructured\_to\_structured() (trong mô-đun numpy.lib.recfunctions), 78