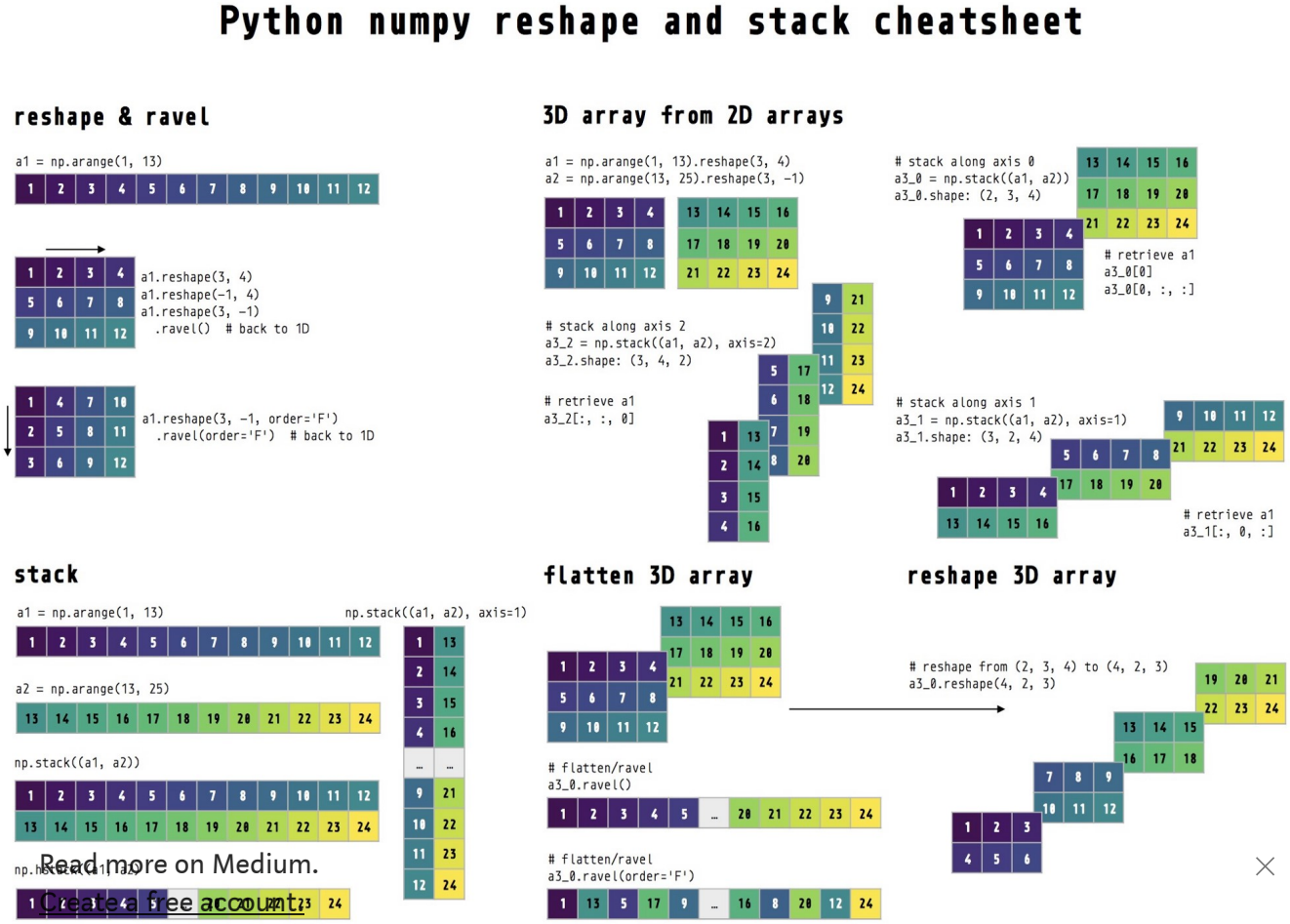
CHEATSHEET AND TUTORIAL FOR NUMPY RESHAPE AND STACK

Reshaping numpy arrays in Python — a step-by-step pictorial tutorial

This tutorial and cheatsheet provide visualizations to help you understand how numpy reshapes multidimensional arrays.

Hause Follow
Dec 25, 2019 · 8 min read ★



Cheatsheet for Python numpy reshape, stack, and flatten (created by Hause Lin and available here)

How does the *numpy* `reshape()` method reshape arrays? Have you been confused or have you struggled understanding how it works? This tutorial will walk you through reshaping in *numpy*. If you want a pdf copy of the cheatsheet above, you can download it here.

Create a Python numpy array

Use `np.arange()` to generate a *numpy* array containing a sequence of numbers from 1 to 12. See documentation here.

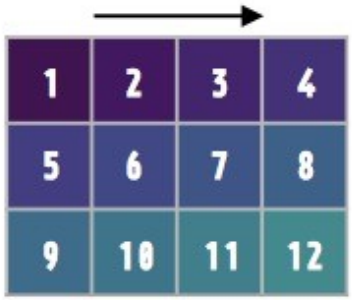
```
import numpy as np

a1 = np.arange(1, 13)  # numbers 1 to 12

print(a1.shape)
> (12,)

print(a1)
> [ 1  2  3  4  5  6  7  8  9 10 11 12]
```

Reshape with reshape() method

Use `reshape()` method to reshape our *a1* array to a 3 by 4 dimensional array. Let's use **3_4** to refer to it dimensions: 3 is the 0th dimension (axis) and 4 is the 1st dimension (axis) (note that Python indexing begins at 0). See documentation here.


```
a1_2d = a1.reshape(3, 4)  # 3_4
```

```
print(a1_2d.shape)
> (3, 4)

print(a1_2d)
> [[ 1  2  3  4]
   [ 5  6  7  8]
   [ 9 10 11 12]]
```



If you want *numpy* to automatically determine what size/length a particular dimension should be, specify the dimension as -1 for that dimension.

```
a1.reshape(3, 4)
a1.reshape(-1, 4)  # same as above: a1.reshape(3, 4)

a1.reshape(3, 4)
a1.reshape(3, -1)  # same as above: a1.reshape(3, 4)

a1.reshape(2, 6)
a1.reshape(2, -1)  # same as above: a1.reshape(2, 6)
```

Reshape along different dimensions

By default, `reshape()` reshapes the array **along the 0th dimension (row)**. This behavior can be changed via the `order` parameter (default value is `'C'`). See documentation for more information.

```
a1.reshape(3, 4) # reshapes or 'fills in' row by row
a1.reshape(3, 4, order='C') # same results as above
```

We can **reshape along the 1st dimension (column)** by changing `order` to `'F'`. For those familiar with MATLAB, MATLAB uses this order.

```
a1.reshape(3, 4, order='F')  # reshapes column by column
> [[ 1  4  7 10]
   [ 2  5  8 11]
   [ 3  6  9 12]]
```

Test: What's the dimension/shape of *array a1*?

a1 is a 1D array — it has only 1 dimension, even though you might think it's dimension should be **1_12** (1 row by 12 columns). To convert to a 1_12 array, use `reshape()`.

```
print(a1)  # what's the shape?
> [ 1  2  3  4  5  6  7  8  9 10 11 12]

print(a1.shape)
> (12,)

a1_1_by_12 = a1.reshape(1, -1)  # reshape to 1_12

print(a1_1_by_12)  # note the double square brackets!
> [[ 1  2  3  4  5  6  7  8  9 10 11 12]]

print(a1_1_by_12.shape)  # 1_12 array
> (1, 12)
```

Flatten/ravel to 1D arrays with ravel()

The `ravel()` method lets you convert multi-dimensional arrays to 1D arrays (see docs here). Our 2D array (**3_4**) will be flattened or raveled such that they become a 1D array with 12 elements.

If you don't specify any parameters, `ravel()` will flatten/ravel our 2D array along the rows (0th dimension/axis). That is, row 0 [1, 2, 3, 4] + row 1 [5, 6, 7, 8] + row 2 [9, 10, 11, 12].

If you want to flatten/ravel along the columns (1st dimension), use the `order` parameter.

```
print(a1_2d)  # 3_4
> [[ 1  2  3  4]
   [ 5  6  7  8]
   [ 9 10 11 12]]

print(a1_2d.ravel())  # ravel by row (default order='C')
> [ 1  2  3  4  5  6  7  8  9 10 11 12]

print(a1_2d.ravel(order='F'))  # ravel by column
> [ 1  5  9  2  6 10  3  7 11  4  8 12]
```

Concatenate/stack arrays with `np.stack()` and `np.hstack()`

Create two 1D arrays

```
a1 = np.arange(1, 13)
print(a1)
> [ 1  2  3  4  5  6  7  8  9 10 11 12]

a2 = np.arange(13, 25)
print(a2)

> [13 14 15 16 17 18 19 20 21 22 23 24]
```

Use `np.stack()` to concatenate/stack arrays. By default, `np.stack()` stacks arrays along the 0th dimension (rows) (parameter `axis=0`). See docs for more info.

```
stack0 = np.stack((a1, a1, a2, a2))  # default stack along 0th
axis
print(stack0.shape)
> (4, 12)

print(stack0)
> [[ 1  2  3  4  5  6  7  8  9 10 11 12]
   [ 1  2  3  4  5  6  7  8  9 10 11 12]
   [13 14 15 16 17 18 19 20 21 22 23 24]
   [13 14 15 16 17 18 19 20 21 22 23 24]]
```

Stack along the 1st dimension (`axis=1`)

```
stack1 = np.stack((a1, a1, a2, a2), axis=1)
print(stack1.shape)
> (12, 4)

print(stack1)
> [[ 1  1 13 13]
   [ 2  2 14 14]
   [ 3  3 15 15]
   [ 4  4 16 16]
   [ 5  5 17 17]
   [ 6  6 18 18]
   [ 7  7 19 19]
   [ 8  8 20 20]
   [ 9  9 21 21]
   [10 10 22 22]
   [11 11 23 23]
   [12 12 24 24]]
```

Concatenate as a long 1D array with `np.hstack()` (stack horizontally)

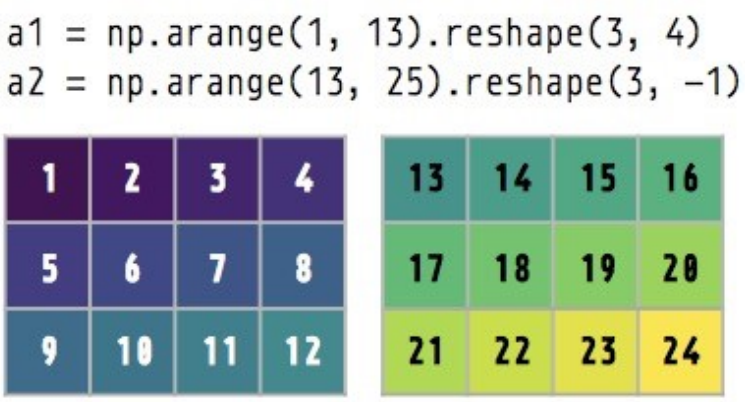
```
stack_long = np.hstack((a1, a2))
print(stack_long.shape)
> (24,)

print(stack_long)
> [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24]
```

Create multi-dimensional array (3D)

Multi-dimensional arrays are very common and are known as tensors. They're used a lot in deep learning and neural networks. If you're into deep learning, you'll be reshaping tensors or multi-dimensional arrays regularly.

Let's begin by first create two different 3 by 4 arrays. We'll combine them to form a 3D array later.

```
a1 = np.arange(1, 13).reshape(3, -1)  # 3_4
a2 = np.arange(13, 25).reshape(3, -1)  # 3_4

print(a1)
> [[ 1  2  3  4]
   [ 5  6  7  8]
   [ 9 10 11 12]]

print(a2)
> [[13 14 15 16]
   [17 18 19 20]
   [21 22 23 24]]
```

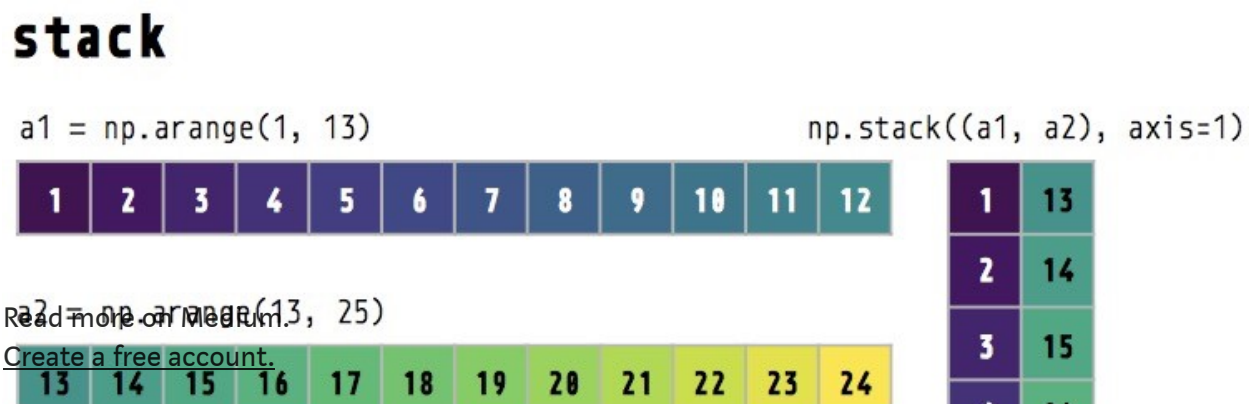


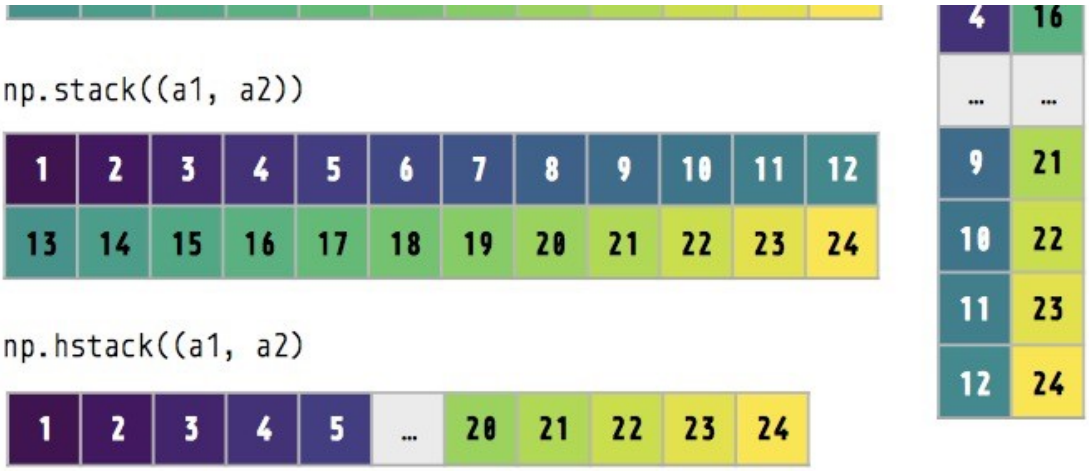
Create a 3D array by stacking the arrays along different axes/dimensions

```
a3_0 = np.stack((a1, a2))  # default axis=0 (dimension 0)
a3_1 = np.stack((a1, a2), axis=1)  # along dimension 1
a3_2 = np.stack((a1, a2), axis=2)  # along dimension 2

print(a3_0.shape)
> (2, 3, 4)
print(a3_1.shape)
> (3, 2, 4)
print(a3_2.shape)
> (3, 4, 2)
```

Let’s print the arrays to see how they look like. See the figure above for visualizations.

```
print(a3_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]]]

print(a3_1)
> [[[ 1  2  3  4]
```





Because the three 3D arrays have been created by stacking two arrays along different dimensions, if we want to retrieve the original two arrays from these 3D arrays, we'll have to subset along the correct dimension/axis.

Test: How can we retrieve our `a1` array from these 3D arrays?

```
print(a1)  # check what's a1
> [[ 1  2  3  4]
   [ 5  6  7  8]
   [ 9 10 11 12]]

# solutions
a3_0[0, :, :]
a3_0[0]  # same as above

a3_1[:, 0, :]

a3_2[:, :, 0]
```

Flatten multidimensional arrays

We can also flatten multi-dimensional arrays with `ravel()`. Below, we ravel row by row (default `order='C'`) to 1D array.

```
print(a3_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]]]

print(a3_0.ravel())
> [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24]
```

Ravel column by column (`order='F'`) to 1D array

```
print(a3_0.ravel(order='F'))
> [ 1 13  5 17  9 21  2 14  6 18 10 22  3 15  7 19 11 23  4 16
8 20 12 24]
```

Reshape multidimensional arrays

We can also use `reshape()` to reshape multi-dimensional arrays.


```
# reshape row by row (default order=C) to 2D array
```

```
print(a3_0)  # 2_3_4
> [[[ 1  2  3  4]
    [ 5  6  7  8]
    [ 9 10 11 12]]

   [[13 14 15 16]
    [17 18 19 20]
    [21 22 23 24]]]

print(a3_0.reshape(4, -1))  # reshape to 4_6 (row by row)
> [[ 1  2  3  4  5  6]
   [ 7  8  9 10 11 12]
   [13 14 15 16 17 18]
   [19 20 21 22 23 24]]

print(a3_0.reshape(4, -1, order='F'))  # reshape (column by
column)
> [[ 1  9  6  3 11  8]
   [13 21 18 15 23 20]
   [ 5  2 10  7  4 12]
   [17 14 22 19 16 24]]

print(a3_0.reshape(4, 2, 3))  # reshape to 4_2_3 (row by row)
> [[[ 1  2  3]
    [ 4  5  6]]

   [[ 7  8  9]
    [10 11 12]]

   [[13 14 15]
    [16 17 18]]

   [[19 20 21]
    [22 23 24]]]
```

Final remarks

I hope now you have a better understanding of how *numpy* reshapes multi-dimensional arrays. I look forward to your thoughts and comments. Also, check out this visual introduction to numpy and data representation.

If you want a pdf copy of the cheatsheet above, you can download it here. If you find this post useful, follow me and visit my site for more data science tutorials.

If you are interested in improving your data science skills, the following articles might be useful:

Two Simple Ways to Loop More Effectively in Python

Use enumerate and zip to write better Python loops

towardsdatascience.com

Code and Develop More Productively With Terminal Multiplexer tmux

Simple tmux commands to improve your productivity

medium.com

Free Online Data Science Courses During COVID-19 Crisis

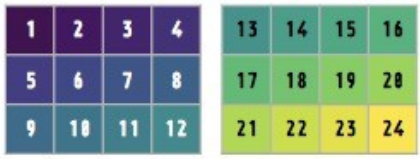
Platforms like Udacity, Codecademy, and Dataquest are now offering their courses for free

towardsdatascience.com

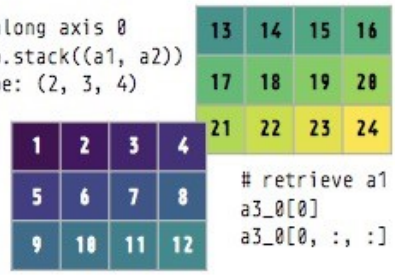
Programming Python Data Science Numpy Towards Data Science

3D array from 2D arrays

```
a1 = np.arange(1, 13).reshape(3, 4)
a2 = np.arange(13, 25).reshape(3, -1)
```

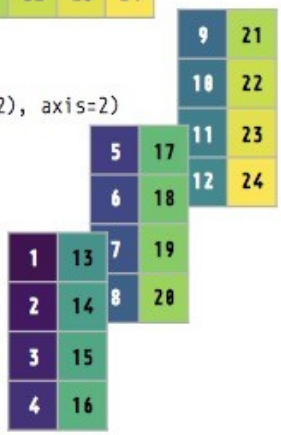



```
# stack along axis 0
a3_0 = np.stack((a1, a2))
a3_0.shape: (2, 3, 4)
```

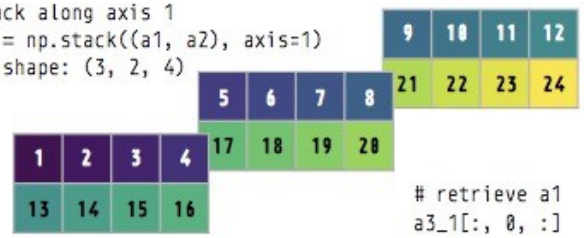



```
# retrieve a1
a3_0[0]
a3_0[0, :, :]
```

```
# stack along axis 2
a3_2 = np.stack((a1, a2), axis=2)
a3_2.shape: (3, 4, 2)

# retrieve a1
a3_2[:, :, 0]
```

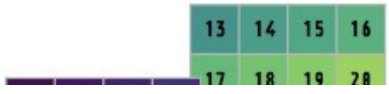



```
# stack along axis 1
a3_1 = np.stack((a1, a2), axis=1)
a3_1.shape: (3, 2, 4)
```

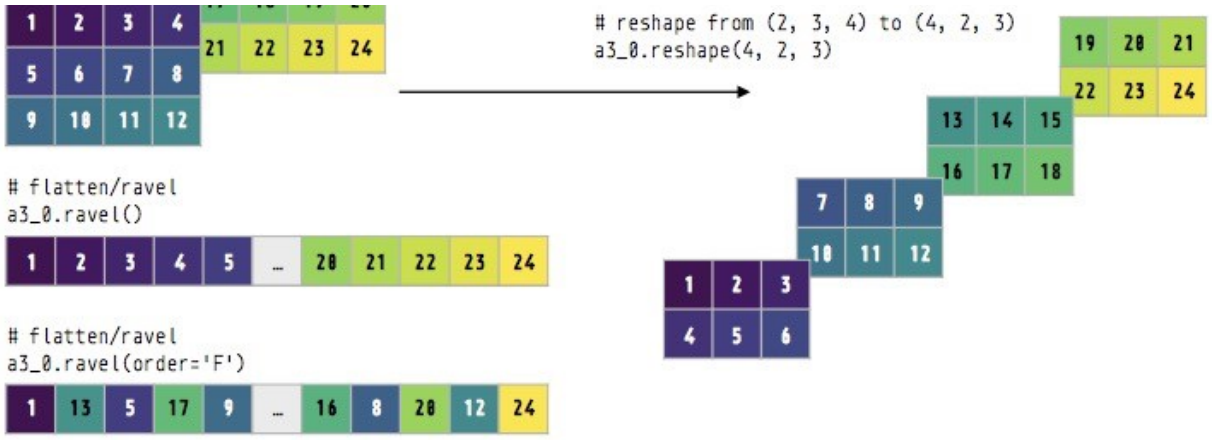



```
# retrieve a1
a3_1[:, 0, :]
```

flatten 3D array



reshape 3D array


1 2 3 4
5 6 7 8
9 10 11 12
21 22 23 24
# reshape from (2, 3, 4) to (4, 2, 3)
a3_0.reshape(4, 2, 3)
19 20 21
22 23 24
13 14 15
16 17 18
7 8 9
10 11 12
1 2 3
4 5 6
# flatten/ravel
a3_0.ravel()
1 2 3 4 5 … 20 21 22 23 24
# flatten/ravel
a3_0.ravel(order='F')
1 13 5 17 9 … 16 8 20 12 24