

Difference between numpy.array shape (R, 1) and (R,)

Asked 10 years, 9 months ago Modified 2 years, 9 months ago

Viewed 244k times



436



In `numpy`, some of the operations return in shape `(R, 1)` but some return `(R,)`. This will make matrix multiplication more tedious since explicit `reshape` is required. For example, given a matrix `M`, if we want to do `numpy.dot(M[:,0], numpy.ones((1, R)))` where `R` is the number of rows (of course, the same issue also occurs column-wise). We will get `matrices are not aligned` error since `M[:,0]` is in shape `(R,)` but `numpy.ones((1, R))` is in shape `(1, R)`.

So my questions are:

1. What's the difference between shape `(R, 1)` and `(R,)`. I know literally it's list of numbers and list of lists where all list contains only a number. Just wondering why not design `numpy` so that it favors shape `(R, 1)` instead of `(R,)` for easier matrix multiplication.
2. Are there better ways for the above example? Without explicitly reshape like this:

```
numpy.dot(M[:,0].reshape(R, 1), numpy.ones((1, R)))
```

[python](#)[numpy](#)[matrix](#)[multidimensional-array](#)[Share](#)[Improve this question](#)[Follow](#)

edited Jan 9, 2015 at 18:06

[Dan D.](#)

74.6k ● 15 ● 110 ● 127

asked Feb 26, 2014 at 20:55

[clwen](#)

20.8k ● 32 ● 82 ● 96

4 [This](#) might help. Not with finding a practical solution though.

– [keyser](#) Feb 26, 2014 at 21:15

2 Proper solution: `numpy.ravel(M[: , 0])` -- converts shape from (R, 1) to (R,) – [Andy R](#) Dec 16, 2017 at 18:38

1 A tuple is not determined by the parentheses, they are not part of it, but by the comma. `x=4,` assigns a tuple, `x=(4)` assigns an int, creating a usual confusion. Shape `n,` expresses the shape of a 1D array with n items, and `n, 1` the shape of a n-row x 1-column array. `(R,)` and `(R, 1)` just add (useless) parentheses but still express respectively 1D and 2D array shapes, Parentheses around a tuple force the evaluation order and prevent it to be read as a list of values (e.g. in function calls). This tuple oddity in mind, things get clearer, NumPy returns the shape which makes sense.

– [mins](#) Oct 24, 2020 at 13:46

8 Answers

Sorted by:

Highest score (default)



1. The meaning of shapes in NumPy

727



You write, "I know literally it's list of numbers and list of lists where all list contains only a number" but that's a bit of an unhelpful way to think about it.



The best way to think about NumPy arrays is that they consist of two parts, a *data buffer* which is just a block of raw elements, and a *view* which describes how to interpret the data buffer.



For example, if we create an array of 12 integers:

```
>>> a = numpy.arange(12)
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

Then `a` consists of a data buffer, arranged something like this:

0	1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	---	----

and a view which describes how to interpret the data:

```
>>> a.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
>>> a.dtype
dtype('int64')
>>> a.itemsize
8
```

```
>>> a.strides
(8,)
>>> a.shape
(12,)
```

Here the *shape* (12,) means the array is indexed by a single index which runs from 0 to 11. Conceptually, if we label this single index *i*, the array *a* looks like this:

i=	0	1	2	3	4	5	6	7	8	9	10
	0	1	2	3	4	5	6	7	8	9	10

If we [reshape](#) an array, this doesn't change the data buffer. Instead, it creates a new view that describes a different way to interpret the data. So after:

```
>>> b = a.reshape((3, 4))
```

the array *b* has the same data buffer as *a*, but now it is indexed by *two* indices which run from 0 to 2 and 0 to 3 respectively. If we label the two indices *i* and *j*, the array *b* looks like this:

i=	0	0	0	0	1	1	1	1	2	2	2
j=	0	1	2	3	0	1	2	3	0	1	2
	0	1	2	3	4	5	6	7	8	9	10

which means that:

```
>>> b[2,1]
9
```

You can see that the second index changes quickly and the first index changes slowly. If you prefer this to be the other way round, you can specify the `order` parameter:

```
>>> c = a.reshape((3, 4), order='F')
```

which results in an array indexed like this:

i=	0	1	2	0	1	2	0	1	2	0	1
j=	0	0	0	1	1	1	2	2	2	3	3
	0	1	2	3	4	5	6	7	8	9	10

which means that:

```
>>> c[2,1]
5
```

It should now be clear what it means for an array to have a shape with one or more dimensions of size 1. After:

```
>>> d = a.reshape((12, 1))
```

the array `d` is indexed by two indices, the first of which runs from 0 to 11, and the second index is always 0:

i=	0	1	2	3	4	5	6	7	8	9	10
j=	0	0	0	0	0	0	0	0	0	0	0

0	1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	---	----

and so:

```
>>> d[10,0]
10
```

A dimension of length 1 is "free" (in some sense), so there's nothing stopping you from going to town:

```
>>> e = a.reshape((1, 2, 1, 6, 1))
```

giving an array indexed like this:

i=	0	0	0	0	0	0	0	0	0	0	0
j=	0	0	0	0	0	0	1	1	1	1	1
k=	0	0	0	0	0	0	0	0	0	0	0
l=	0	1	2	3	4	5	0	1	2	3	4
m=	0	0	0	0	0	0	0	0	0	0	0

0	1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	---	----

and so:

```
>>> e[0,1,0,0,0]
6
```

See the [NumPy internals documentation](#) for more details about how arrays are implemented.

2. What to do?

Since `numpy.reshape` just creates a new view, you shouldn't be scared about using it whenever necessary. It's the right tool to use when you want to index an array in a different way.

However, in a long computation it's usually possible to arrange to construct arrays with the "right" shape in the first place, and so minimize the number of reshapes and transposes. But without seeing the actual context that led to the need for a reshape, it's hard to say what should be changed.

The example in your question is:

```
numpy.dot(M[:,0], numpy.ones((1, R)))
```

but this is not realistic. First, this expression:

```
M[:,0].sum()
```

computes the result more simply. Second, is there really something special about column 0? Perhaps what you actually need is:

```
M.sum(axis=0)
```

answered Feb 27, 2014 at 16:26

**Gareth Rees**


65.8k ● 10 ● 137 ● 165

52 This was extremely helpful in thinking about how arrays are stored. Thank you! Accessing a column (or row) of a (2-d) matrix for further matrix computation is inconvenient though since I always have to reshape the column appropriately. Everytime I have to change the shape from (n,) to (n,1).

– [OfLettersAndNumbers](#) May 30, 2016 at 18:35

4 @SammyLee: Use [newaxis](#) if you need another axis, for example `a[:, j, np.newaxis]` is the `j` th column of `a`, and `a[np.newaxis, i]` is the `i` th row. – [Gareth Rees](#) May 30, 2016 at 18:47

1 i am trying to plot indices to get a better understanding on paper by this model and I dont seem to get it, if I had a shape 2 x 2 x 4 i understand the first 2 can be understood as 0000000011111111 and the last 4 can be understood as 0123012301230123 what happens to the middle one? – [PirateApp](#) Apr 20, 2018 at 16:07

5 An easy way to think of this is that numpy is working exactly as expected here, but Python's printing of tuples can be misleading. In the `(R,)` case, the shape of the `ndarray` is a tuple with a single elements, so is printed by Python with a trailing comma. Without the extra comma, it would be [ambiguous with an expression in parenthesis](#). A `ndarray` with a single dimension can be thought of as a column vector of length `R`. In the `(R, 1)` case, the tuple has two elements, so can be thought of as a row vector (or a matrix with 1 row of length `R`). – [Michael Yang](#) Apr 13, 2019 at 21:06 

1 @Alex-droidAD: See [this question](#) and its answers.
– [Gareth Rees](#) May 10, 2019 at 11:35



30



The difference between `(R,)` and `(1, R)` is literally the number of indices that you need to use. `ones((1, R))` is a 2-D array that happens to have only one row. `ones(R)` is a vector. Generally if it doesn't make sense for the variable to have more than one row/column, you should be using a vector, not a matrix with a singleton dimension.

For your specific case, there are a couple of options:

1) Just make the second argument a vector. The following works fine:

```
np.dot(M[:, 0], np.ones(R))
```

2) If you want matlab like matrix operations, use the class `matrix` instead of `ndarray`. All matrices are forced into being 2-D arrays, and operator `*` does matrix multiplication instead of element-wise (so you don't need `dot`). In my experience, this is more trouble than it is worth, but it may be nice if you are used to matlab.

Share Improve this answer

edited Nov 6, 2017 at 9:29

Follow



Eypros

5,703 ● 6 ● 51 ● 85

answered Feb 26, 2014 at 21:30



Evan

2,347 ● 17 ● 19

-
- 1 Yes. I was expecting a more matlab-like behavior. I'll take a look at `matrix` class. What's the trouble for `matrix` class BTW? – [clwen](#) Feb 26, 2014 at 22:18 ✎
-
- 4 The problem with `matrix` is that it is only 2D, and also that because it overloads operator `*`, functions written for `ndarray` may fail if used on a `matrix`. – [Evan](#) Feb 26, 2014 at 22:58 ✎
-



25



The shape is a tuple. If there is only 1 dimension the shape will be one number and just blank after a comma. For 2+ dimensions, there will be a number after all the commas.

```
# 1 dimension with 2 elements, shape = (2,).
# Note there's nothing after the comma.
z=np.array([ # start dimension
```



```
    10,      # not a dimension
    20      # not a dimension
])         # end dimension
print(z.shape)
```

(2,)

```
# 2 dimensions, each with 1 element, shape = (2,1)
w=np.array([ # start outer dimension
    [10],    # element is in an inner dimension
    [20]     # element is in an inner dimension
])         # end outer dimension
print(w.shape)
```

(2,1)

Share Improve this answer

edited Mar 11, 2018 at 14:23

Follow

answered Mar 11, 2018 at 14:04



Katie Jergens

351 ● 3 ● 4

-
- 1 Classic. So many complicated answers, and then way down I find this which exactly explains it. Thank you! – [Michael](#) Dec 17, 2021 at 3:29
-



For its base array class, 2d arrays are no more special than 1d or 3d ones. There are some operations that

5

preserve the dimensions, some that reduce them, other combine or even expand them.



```
M=np.arange(9).reshape(3,3)
M[:,0].shape # (3,) selects one column, returns a 1d a
M[0,:].shape # same, one row, 1d array
M[:,[0]].shape # (3,1), index with a list (or array),
M[:,[0,1]].shape # (3,2)
```

```
In [20]: np.dot(M[:,0].reshape(3,1),np.ones((1,3)))
```

```
Out[20]:
array([[ 0.,  0.,  0.],
       [ 3.,  3.,  3.],
       [ 6.,  6.,  6.]])
```

```
In [21]: np.dot(M[:,[0]],np.ones((1,3)))
```

```
Out[21]:
array([[ 0.,  0.,  0.],
       [ 3.,  3.,  3.],
       [ 6.,  6.,  6.]])
```

Other expressions that give the same array

```
np.dot(M[:,0][:,np.newaxis],np.ones((1,3)))
np.dot(np.atleast_2d(M[:,0]).T,np.ones((1,3)))
np.einsum('i,j',M[:,0],np.ones((3)))
M1=M[:,0]; R=np.ones((3)); np.dot(M1[:,None], R[None,:])
```

MATLAB started out with just 2D arrays. Newer versions allow more dimensions, but retain the lower bound of 2. But you still have to pay attention to the difference between a row matrix and column one, one with shape $(1,3)$ v $(3,1)$. How often have you written $[1,2,3].'$? I was going to write **row vector** and **column vector**, but with that 2d constraint, there aren't any vectors in

MATLAB - at least not in the mathematical sense of vector as being 1d.

Have you looked at `np.atleast_2d` (also `_1d` and `_3d` versions)?

In newer Python/numpy there's a `matmul` operator

```
In [358]: M[:,0,np.newaxis]@np.ones((1,3))
Out[358]:
array([[0., 0., 0.],
       [3., 3., 3.],
       [6., 6., 6.]])
```

In `numpy` element-wise multiplication is in a sense more basic than matrix multiplication. With the sum-of-products on a size 1 dimension, there's no need to use `dot/matmul`:

```
In [360]: M[:,0,np.newaxis]*np.ones((1,3))
Out[360]:
array([[0., 0., 0.],
       [3., 3., 3.],
       [6., 6., 6.]])
```

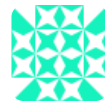
This uses `broadcasting`, a powerful feature that `numpy` has had all along. MATLAB only added it recently.

Share Improve this answer

[edited Apr 26, 2021 at 16:31](#)

Follow

answered Feb 27, 2014 at 3:10



hpaulj

231k ● 14 ● 254 ● 377



3

There are a lot of good answers here already. But for me it was hard to find some example, where the shape or array can break all the program.



So here is the one:



```
import numpy as np
a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])
```

```
from sklearn.linear_model import LinearRegression
regr = LinearRegression()
regr.fit(a, b)
```

This will fail with error:

ValueError: Expected 2D array, got 1D array instead

but if we add `reshape` to `a`:

```
a = np.array([1, 2, 3, 4]).reshape(-1, 1)
```

this works correctly!

answered Dec 18, 2019 at 13:16

Share Improve this answer

Follow



Mikhail_Sam

11.2k ● 11 ● 76 ● 110

-
- 1 Also TensorFlow 2.4 see e.g.
stackoverflow.com/questions/67662727/... – Julian Moore
May 24, 2021 at 15:40
-



3



The data structure of shape $(n,)$ is called a rank 1 array. It doesn't behave consistently as a row vector or a column vector which makes some of its operations and effects non intuitive. If you take the transpose of this $(n,)$ data structure, it'll look exactly same and the dot product will give you a number and not a matrix. The vectors of shape $(n,1)$ or $(1,n)$ row or column vectors are much more intuitive and consistent.

Share Improve this answer

Follow

answered Dec 4, 2020 at 10:47



Palak Bansal

840 ● 12 ● 26

-
- 1 Your intuition has been shaped by linear algebra and/or MATLAB like languages, ones that work primarily with 2d arrays, matrices. In MATLAB everything is 2d, even "scalars". We use Python and `numpy` for a lot more than `dot` products:) – hpaulj Apr 26, 2021 at 16:22
-

I agree. Dot product helped me understand the structure better, I have mentioned it for the same reason :)

– Palak Bansal Apr 27, 2021 at 10:22



1



1) The reason not to prefer a shape of `(R, 1)` over `(R,)` is that it unnecessarily complicates things. Besides, why would it be preferable to have shape `(R, 1)` by default for a length-R vector instead of `(1, R)`? It's better to keep it simple and be explicit when you require additional dimensions.

2) For your example, you are computing an outer product so you can do this without a `reshape` call by using `np.outer`:

```
np.outer(M[:,0], numpy.ones((1, R)))
```

Share Improve this answer

edited Feb 26, 2014 at 21:56

Follow

answered Feb 26, 2014 at 21:46



[bogatron](#)

19.1k ● 7 ● 55 ● 49

Thanks for the answer. 1) `M[:,0]` is essentially getting all rows with first element, so it makes more sense to have `(R, 1)` than `(1, R)`. 2) It's not always replaceable by `np.outer`, e.g., dot for matrix in shape `(1, R)` then `(R, 1)`.
– [clwen](#) Feb 26, 2014 at 22:17

1) Yes, that *could* be the convention but that makes it less convenient in other circumstances. The convention could also be for `M[1, 1]` to return a shape `(1, 1)` array but that is also usually less convenient than a scalar. If you really want matrix-like behaviour, then you would be better of using a `matrix` object. 2) Actually, `np.outer` works regardless of

whether the shapes are `(1, R)`, `(R, 1)`, or a combination of the two. – [bogatron](#) Feb 27, 2014 at 14:26



1



To be clear, we are talking about:

- a NumPy array also known as `numpy.ndarray`
- the shape of an array known by `numpy.ndarray.shape`
- the question assumes some unknown `numpy.ndarray` with the shape `(R,)` where `R` should be understood as the length of its respective dimension

NumPy arrays have a shape. That `.shape` is represented by a tuple where each element in the tuple tells us the length of that dimension. To keep it simple, let's stick to rows and columns. While the values of our `numpy.ndarray` will not change in the following examples, the shape will.

Let's consider an array with the values 1, 2, 3, and 4.

Our examples will include the following `.shape` representations:

```
(4,) # 1-dimensional array with length 4
(1,4) # 2-dimensional array with row length 1, column
(4,1) # 2-dimensional array with row length 4, column
```

We can think of this more abstractly with variables `a` and `b`.

```
(a,) # 1-dimensional array with length a
(b,a) # 2-dimensional array with row length b, column
(a,b) # 2-dimensional array with row length a, column
```

For me, it is helpful to 'manually' build these out to get a better feel for what their dimensions mean.

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

>> # (1,4)
>> row_vector = np.array(
    [
        [1, 2, 3, 4]
    ]
)

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

So, the answer to the first question:

1. What's the difference between shape `(R, 1)` and `(R,)`?

Answer: They have different dimensions. `a` is the length of the one dimension and `b` the length of another, `.shape` is `(a, b)` and `(a,)` respectively. `b` just happens to be 1. One way to think of this is if `a = 1` then the row has length 1 thus it is a row vector. If `b = 1` then the column has length 1 so the `numpy.ndarray` it represents is a column vector.

2. Are there better ways for the above example?

Answer: Let's assume we have the array I used as example above with 1, 2, 3, and 4 as values. A convenient way to get `(R,)` to be `(R, 1)` is this:

```
>> one_dimensional_array = np.array([1, 2, 3, 4])
>> one_dimensional_array.shape
(4,)
>> row_vector = one_dimensional_array[:, None]
>> row_vector.shape
(4, 1)
```

Resources

1. NumPy — ndarrays —

<https://numpy.org/doc/stable/reference/arrays.ndarray.html>

2. Cross Validated @unutbu — dimension trick —

<https://stats.stackexchange.com/a/285005>

Share Improve this answer

answered Mar 11, 2022 at 4:16

Follow



Jesse H.

725 ● 7 ● 12
