A Brief Introduction to Python
Part II: Numpy

Wei Tianwen

2017

Table of contents

Introduction

- Native Python does not provide an appropriate class for manipulating higher-dimensional arrays, e.g. matrices, This is an elementary needs in scientific computing and many other application areas.
- A powerful third-party Python package called Numpy filled this blank. Thanks to the great effort of open source community, poor people like us do not need to pirate Matlab anymore.
- In this tutorial, we are going to learn the `ndarray` class along with many useful functions provided by Numpy.

A first example

Let us begin with the simplest example as always:

```
>>> import numpy as np
>>> a = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> type(a)
<class 'list'>
>>> a
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> arr = np.array(a)
>>> type(arr)
<class 'numpy.ndarray'>
>>> arr
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

A first example

Let us begin with the simplest example as always:

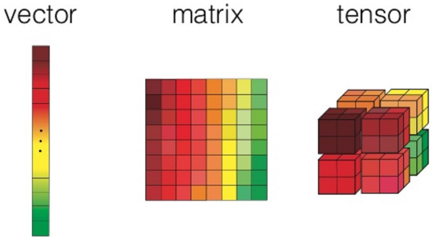
```
>>> import numpy as np
>>> a = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> type(a)
<class 'list'>
>>> a
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> arr = np.array(a)
>>> type(arr)
<class 'numpy.ndarray'>
>>> arr
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Note that `np.array()` accepts a `list` or `tuple` object as input. The following code does not work:

```
>>> b = np.array(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: only 2 non-keyword arguments accepted
```

N-dimensional array

The class name `ndarray` is an abbreviation for "*n*-dimensional array". So what is exactly an "*n*-dimensional array"? Basically,



- Vector is 1-dimensional, i.e. $\mathbf{v} = (v_i) \in \mathbb{R}^n$.
- Matrix is 2-dimensional, i.e. $\mathbf{M} = (M_{ij}) \in \mathbb{R}^{n_1 \times n_2}$
- Tensor is 3-dimensional or higher, i.e. $\mathcal{T} = (T_{ijk}) \in \mathbb{R}^{n_1 \times n_2 \times n_3}$.

To conclude, a general m-dimensional array looks like:

$$\mathcal{T} = (T_{i_1,i_2,\ldots,i_m}) \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_m}.$$

Exercise 1.1

How many elements does $\mathcal{T}$ have as described above?

To conclude, a general m-dimensional array looks like:

$$\mathcal{T} = (T_{i_1,i_2,\dots,i_m}) \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_m}.$$

Exercise 1.1

How many elements does $\mathcal{T}$ have as described above?

Solution: Since each index i_p varies from 1 to n_p, array $\mathcal{T}$ must have $n_1 \times n_2 \times \cdots \times n_m$ elements.

Note that in most programming languages such as C and Python, the convention is that each index i_p varies from 0 to $n_p - 1$ instead.

Function `np.array()` provided by `numpy` creates a `ndarray` object from a `list`. More specifically,

- It creates a 1-dimensional array from a list:

```
>>> a1 = [1, 2, 3]
>>> array1 = np.array(a1)
>>> array1
array([1, 2, 3])
```

- It creates a 2-dimensional array from a list of lists:

```
>>> b1 = [4, 5, 6]
>>> a2 = [a1, b1]  # a1 and b1 are lists
>>> a2
[[1, 2, 3], [4, 5, 6]]
>>> array2 = np.array(a2)
>>> array2
array([[1, 2, 3],
       [4, 5, 6]])
```

- It creates a 3-dimensional array from a list of lists of lists:

```
>>> b2 = [[7, 8, 9], [10,11,12]]
>>> a3 = [a2, b2] # a2 and b2 are lists of lists
>>> a3
[[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]]
>>> array3 = np.array(a3)
>>> array3
```

Attributes

Below are some of the most important attributes of the `ndarray` class:

- `ndarray.ndim`: the number of axes (dimensions) of the array, also known as the **rank**. It is the number n in the n-dimensional array.

```
>>> array1.ndim  # vector has rank 1
1
>>> array2.ndim  # matrix has rank 2
2
>>> array3.ndim  # higher-dimensional matrix has rank 3 or above
3
```

Attributes

Below are some of the most important attributes of the `ndarray` class:

- `ndarray.ndim` : the number of axes (dimensions) of the array, also known as the **rank**. It is the number *n* in the *n*-dimensional array.

```
>>> array1.ndim  # vector has rank 1
1
>>> array2.ndim  # matrix has rank 2
2
>>> array3.ndim  # higher-dimensional matrix has rank 3 or above
3
```

- `ndarray.shape` : the dimensions of the array. For a matrix with n rows and m columns, its shape will be `(n,m)` .

```
>>> array1.shape
(3,)
>>> array2.shape
(2, 3)
>>> array3.shape
(2, 2, 3)
```

Note that we always have `arr.ndim == len(arr.shape)`

Attributes

- `ndarray.size` : the total number of elements in the array.

```
>>> array1.size
3
>>> array2.size
6
>>> array3.size
12
```

Attributes

- `ndarray.size`: the total number of elements in the array.

```
>>> array1.size
3
>>> array2.size
6
>>> array3.size
12
```

- `ndarray.dtype`: describing the type of the elements in the array. One can create or specify dtype's using standard Python types such as `float` or types provided by Numpy, e.g. `numpy.int32`.

```
>>> array1.dtype
dtype('int32')
>>> array4 = np.array([1.0, 2.5])
>>> array4.dtype
dtype('float64')
```

More about creating an array

Typically, `np.array()` use the `dtype` of the input for the output array. For instance:

```
>>> a = np.array([1,2,3])
>>> a  # an array of integers by default
array([1, 2, 3])
>>> b = np.array([1.1, 2.5, 3.3])
>>> b  # an array of floating points by default
array([ 1.1,  2.5,  3.3])
```

But we can also specify the `dtype` argument when calling `np.array()` to force the type conversion.

```
>>> a = np.array([1,2,3], dtype=float)
>>> a  # an array of floating points
array([ 1.,  2.,  3.])
>>> b = np.array([1.1, 2.5, 3.3], dtype=int)
>>> b  # an array of integers
array([1, 2, 3])
```

Creating an array of zeros and ones

Numpy also provides function `np.zeros()` to create an array full of 0 and `np.ones()` to create an array full of 1.

```
>>> a = np.zeros(shape=(2, 5))
>>> a
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> b = np.ones(shape=(3, 4))
>>> b
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

We observe that an array of floating points is returned by default. To create an integer array, we must specify `dtype` argument:

```
>>> a=np.zeros((2, 5), dtype=int)
>>> a
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
>>> b=np.ones((3, 4), dtype=int)
>>> b
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
```

Shape

Notice that we must specify the `shape` argument when calling `np.zeros()` and `np.ones()`. The `shape` argument accepts a `tuple` (preferred) or `list` object as input.

```
>>> a = np.zeros((2, 5)) # OK, preferred
>>> a = np.zeros([2, 5]) # OK
>>> a = np.zeros(2, 5)  # Error
```

One special case is that when creating a 1-dimensional array, `shape` parameter also accept `int` value:

```
>>> a = np.zeros(3) # it works although 3 is not a tuple
>>> a
array([ 0.,  0.,  0.])
>>> b = np.ones(3)
>>> b
array([ 1.,  1.,  1.])
```

Creating sequences of numbers

To create sequences of numbers, Numpy provides two useful functions: `np.arange()` and `np.linspace()`.

- Function `np.arange()` accepts 3 arguments: `start`, `stop` and `step`:

```
>>> np.arange(start=10, stop=30, step=5 )
array([10, 15, 20, 25])
>>> np.arange(0, 2, 0.3)  # it accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

- Function `np.linspace()` allows the users to specify the number `num` of elements to be created instead of the step size `step`:

```
>>> np.linspace(start=0, stop=2, num=9)  # 9 numbers from 0 to 2
array([0.,  0.25,  0.5,  0.75,  1.,  1.25,  1.5,  1.75,  2. ])
```

Accessing single element of an array

Just like accessing element of a list, the index always starts from 0 and ends at $n-1$.

```
>>> A = array2
>>> A
array([[1, 2, 3],
       [4, 5, 6]])
>>> A[0,0]
1
>>> A[0,1]
2
>>> A[0,2]
3
>>> A[1,0]
4
>>> A[1,1]
5
>>> A[1,2]
6
>>> A[1,-1]  # index can be negative, -1 means the last index
6
>>> A[1,-2]
5
>>> A[1,-3]
4
```

Slicing an array

Let us now talk about how to extract a subarray from a given array. First, we generate an array with 10 rows and 8 columns.

```
>>> A=np.arange(1,81).reshape(10,8)
>>> A
array([[ 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, 30, 31, 32],
       [33, 34, 35, 36, 37, 38, 39, 40],
       [41, 42, 43, 44, 45, 46, 47, 48],
       [49, 50, 51, 52, 53, 54, 55, 56],
       [57, 58, 59, 60, 61, 62, 63, 64],
       [65, 66, 67, 68, 69, 70, 71, 72],
       [73, 74, 75, 76, 77, 78, 79, 80]])
```

	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16
2	17	18	19	20	21	22	23	24
3	25	26	27	28	29	30	31	32
4	33	34	35	36	37	38	39	40
5	41	42	43	44	45	46	47	48
6	49	50	51	52	53	54	55	56
7	57	58	59	60	61	62	63	64
8	65	66	67	68	69	70	71	72
9	73	74	75	76	77	78	79	80

To obtain the subarray covering the blue area, enter

```
>>> A[2:8, 3:7]
array([[20, 21, 22, 23],
       [28, 29, 30, 31],
       [36, 37, 38, 39],
       [44, 45, 46, 47],
       [52, 53, 54, 55],
       [60, 61, 62, 63]])
```

	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16
2	17	18	19	20	21	22	23	24
3	25	26	27	28	29	30	31	32
4	33	34	35	36	37	38	39	40
5	41	42	43	44	45	46	47	48
6	49	50	51	52	53	54	55	56
7	57	58	59	60	61	62	63	64
8	65	66	67	68	69	70	71	72
9	73	74	75	76	77	78	79	80

To obtain the subarray covering the blue area, enter

```
>>> A[4:9, :]  # equivalent to A[4:-1, :]
array([[33, 34, 35, 36, 37, 38, 39, 40],
       [41, 42, 43, 44, 45, 46, 47, 48],
       [49, 50, 51, 52, 53, 54, 55, 56],
       [57, 58, 59, 60, 61, 62, 63, 64],
       [65, 66, 67, 68, 69, 70, 71, 72]])
```

	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16
2	17	18	19	20	21	22	23	24
3	25	26	27	28	29	30	31	32
4	33	34	35	36	37	38	39	40
5	41	42	43	44	45	46	47	48
6	49	50	51	52	53	54	55	56
7	57	58	59	60	61	62	63	64
8	65	66	67	68	69	70	71	72
9	73	74	75	76	77	78	79	80

To obtain the subarray covering the blue area, enter

```
>>> A[:, 4:]
array([[ 5,  6,  7,  8],
       [13, 14, 15, 16],
       [21, 22, 23, 24],
       [29, 30, 31, 32],
       [37, 38, 39, 40],
       [45, 46, 47, 48],
       [53, 54, 55, 56],
       [61, 62, 63, 64],
       [69, 70, 71, 72],
       [77, 78, 79, 80]])
```

	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16
2	17	18	19	20	21	22	23	24
3	25	26	27	28	29	30	31	32
4	33	34	35	36	37	38	39	40
5	41	42	43	44	45	46	47	48
6	49	50	51	52	53	54	55	56
7	57	58	59	60	61	62	63	64
8	65	66	67	68	69	70	71	72
9	73	74	75	76	77	78	79	80

To obtain the subarray covering the blue area, enter

```
>>> A[:4, :5]  # equivalent to A[0:4, 0:5]
array([[ 1,  2,  3,  4,  5],
       [ 9, 10, 11, 12, 13],
       [17, 18, 19, 20, 21],
       [25, 26, 27, 28, 29]])
```

Exercise

	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16
2	17	18	19	20	21	22	23	24
3	25	26	27	28	29	30	31	32
4	33	34	35	36	37	38	39	40
5	41	42	43	44	45	46	47	48
6	49	50	51	52	53	54	55	56
7	57	58	59	60	61	62	63	64
8	65	66	67	68	69	70	71	72
9	73	74	75	76	77	78	79	80

How to obtain the subarray covering the blue area?

Exercise

	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16
2	17	18	19	20	21	22	23	24
3	25	26	27	28	29	30	31	32
4	33	34	35	36	37	38	39	40
5	41	42	43	44	45	46	47	48
6	49	50	51	52	53	54	55	56
7	57	58	59	60	61	62	63	64
8	65	66	67	68	69	70	71	72
9	73	74	75	76	77	78	79	80

How to obtain the subarray covering the blue area?

Solution:

```
>>> A[7:, 4:]
array([[61, 62, 63, 64],
       [69, 70, 71, 72],
       [77, 78, 79, 80]])
```

Usage of `start:stop:step`

```
>>> C = np.array(np.arange(1,21))
>>> C
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20])
>>> C[0:15]  # equivalent to C[:15]
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> C[3:15]
array([ 4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> C[0:15:2]
array([ 1,  3,  5,  7,  9, 11, 13, 15])
>>> C[0:15:3]
array([ 1,  4,  7, 10, 13])
>>> C[10::3]
array([11, 14, 17, 20])
>>> C[::3]  # equivalent to C[0::3]
array([ 1,  4,  7, 10, 13, 16, 19])
```

Usage of `start:stop:step`

```
>>> C = np.array(np.arange(1,21))
>>> C
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20])
>>> C[0:15]  # equivalent to C[:15]
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> C[3:15]
array([ 4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> C[0:15:2]
array([ 1,  3,  5,  7,  9, 11, 13, 15])
>>> C[0:15:3]
array([ 1,  4,  7, 10, 13])
>>> C[10::3]
array([11, 14, 17, 20])
>>> C[::3]  # equivalent to C[0::3]
array([ 1,  4,  7, 10, 13, 16, 19])
```

We can see that

- if `start` is omitted, then it means "starts from the beginning", i.e. `start=0`;
- if `stop` is omitted, then it means "to (and include) the last element";
- if `step` is omitted, then it means `step=1`.

	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16
2	17	18	19	20	21	22	23	24
3	25	26	27	28	29	30	31	32
4	33	34	35	36	37	38	39	40
5	41	42	43	44	45	46	47	48
6	49	50	51	52	53	54	55	56
7	57	58	59	60	61	62	63	64
8	65	66	67	68	69	70	71	72
9	73	74	75	76	77	78	79	80

To obtain the subarray covering the blue area, enter

```
>>> A[:, 1::2]
array([[ 2,  4,  6,  8],
       [10, 12, 14, 16],
       [18, 20, 22, 24],
       [26, 28, 30, 32],
       [34, 36, 38, 40],
       [42, 44, 46, 48],
       [50, 52, 54, 56],
       [58, 60, 62, 64],
       [66, 68, 70, 72],
       [74, 76, 78, 80]])
```

	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16
2	17	18	19	20	21	22	23	24
3	25	26	27	28	29	30	31	32
4	33	34	35	36	37	38	39	40
5	41	42	43	44	45	46	47	48
6	49	50	51	52	53	54	55	56
7	57	58	59	60	61	62	63	64
8	65	66	67	68	69	70	71	72
9	73	74	75	76	77	78	79	80

How to obtain the subarray covering the blue area?

	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16
2	17	18	19	20	21	22	23	24
3	25	26	27	28	29	30	31	32
4	33	34	35	36	37	38	39	40
5	41	42	43	44	45	46	47	48
6	49	50	51	52	53	54	55	56
7	57	58	59	60	61	62	63	64
8	65	66	67	68	69	70	71	72
9	73	74	75	76	77	78	79	80

How to obtain the subarray covering the blue area?

```
>>> A[0:7:2, :]   # equivalent to A[:7:2, :]
array([[ 1,  2,  3,  4,  5,  6,  7,  8],
       [17, 18, 19, 20, 21, 22, 23, 24],
       [33, 34, 35, 36, 37, 38, 39, 40],
       [49, 50, 51, 52, 53, 54, 55, 56]])
```

Exercise 4.1

Create a 10x10 matrix A using command `A=np.arange(1,101).reshape((10,10))`. This gives

```
>>> A
array([[  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,  30],
       [ 31,  32,  33,  34,  35,  36,  37,  38,  39,  40],
       [ 41,  42,  43,  44,  45,  46,  47,  48,  49,  50],
       [ 51,  52,  53,  54,  55,  56,  57,  58,  59,  60],
       [ 61,  62,  63,  64,  65,  66,  67,  68,  69,  70],
       [ 71,  72,  73,  74,  75,  76,  77,  78,  79,  80],
       [ 81,  82,  83,  84,  85,  86,  87,  88,  89,  90],
       [ 91,  92,  93,  94,  95,  96,  97,  98,  99, 100]])
```

1. Using slicing to create 5×5 matrices $A_{11}, A_{12}, A_{21}, A_{22}$ such that

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}.$$

2. Using slicing to create matrices B_1 and B_2 such that
   - B_1 consists of odd rows of A;
   - B_2 consists of even columns of A.

Solution

```
>>> A11=A[0:5,0:5]
>>> A12=A[0:5,5:10]
>>> A21=A[5:10,0:5]
>>> A22=A[5:10,5:10]
array([[ 56,  57,  58,  59,  60],
       [ 66,  67,  68,  69,  70],
       [ 76,  77,  78,  79,  80],
       [ 86,  87,  88,  89,  90],
       [ 96,  97,  98,  99, 100]])
>>>
>>> B1=A[::2,:]
>>> B1
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10],
       [21, 22, 23, 24, 25, 26, 27, 28, 29, 30],
       [41, 42, 43, 44, 45, 46, 47, 48, 49, 50],
       [61, 62, 63, 64, 65, 66, 67, 68, 69, 70],
       [81, 82, 83, 84, 85, 86, 87, 88, 89, 90]])
>>> B2=A[:,1::2]
>>> B2
array([[  2,   4,   6,   8,  10],
       [ 12,  14,  16,  18,  20],
       [ 22,  24,  26,  28,  30],
       [ 32,  34,  36,  38,  40],
       [ 42,  44,  46,  48,  50],
       [ 52,  54,  56,  58,  60],
       [ 62,  64,  66,  68,  70],
       [ 72,  74,  76,  78,  80],
       [ 82,  84,  86,  88,  90],
       [ 92,  94,  96,  98, 100]])
```

Methods

The `ndarray` class contains a various methods. Below are some of them:

`max()`	return maximum element
`argmax()`	return the index of the maximum element
`dot(y)`	return the matrix multiplication with `y`
`sum()`	return the sum
`mean()`	return the mean
`var()`	return the variance
`ravel()`	return the flattened array
`transpose()`	return the transposed array
`reshape()`	return the reshaped array

Examples

- `max()`

```
>>> x=np.array([[1, 2, 3], [4, 5, 6]])
>>> x.max()
6
>>> x.max(axis=0)
array([4, 5, 6])
>>> x.max(axis=1)
array([3, 6])
```

- `argmax()`

```
>>> x.argmax()
5
>>> x.argmax(axis=0)
array([1, 1, 1], dtype=int64)
>>> x.argmax(axis=1)
array([2, 2], dtype=int64)
```

- `dot()`

```
>>> y = np.array([[1, 3], [2, 4], [5, 6]])
>>> x.dot(y)
array([[20, 29],
       [44, 68]])
```

- `sum`

```
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.sum(axis=0)
array([5, 7, 9])
>>> x.sum(axis=1)
array([ 6, 15])
```

- `mean()`

```
>>> x.mean()
3.5
>>> x.mean(axis=1)
array([ 2.,  5.])
>>> x.mean(axis=0)
array([ 2.5,  3.5,  4.5])
```

- `var()`

```
>>> x.var()
2.9166666666666665
>>> x.var(axis=1)
array([ 0.66666667,  0.66666667])
>>> x.var(axis=0)
array([ 2.25,  2.25,  2.25])
```

- `ravel()`

```
>>> x.ravel()
array([1, 2, 3, 4, 5, 6])
>>> x.ravel()[x.argmax()]
6
```

- `transpose()`

```
>>> x.transpose()
array([[1, 4],
       [2, 5],
       [3, 6]])
```

- `reshape()`

```
>>> x.reshape((1,6))
array([[1, 2, 3, 4, 5, 6]])
>>> x.reshape((6,1))
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
>>> x.reshape((6,))
array([1, 2, 3, 4, 5, 6])
```

Deep and shallow copies of array

We stress that in Numpy assignment gives shallow copy:

```
>>> x=np.array([[1,2,3], [4,5,6]])
>>> y=x
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
>>> x[0,0]=100
>>> y  # y[0,0] changed
array([[100,   2,   3],
       [  4,   5,   6]])
```

Deep and shallow copies of array

We stress that in Numpy assignment gives shallow copy:

```
>>> x=np.array([[1,2,3], [4,5,6]])
>>> y=x
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
>>> x[0,0]=100
>>> y  # y[0,0] changed
array([[100,   2,   3],
       [  4,   5,   6]])
```

To obtain a deep copy, we have to use the `copy()` method:

```
>>> z=x.copy()
>>> z
array([[100,   2,   3],
       [  4,   5,   6]])
>>> x[0,0]=1
>>> z  # z does not change
array([[100,   2,   3],
       [  4,   5,   6]])
```

View of an array

Recall that `ravel()`, `transpose()` and `reshape()` method produce a new array of a different shape.

```
>>> x=np.array([[1,2,3],[4,5,6]])
>>> y1=x.ravel()
>>> y1
array([1, 2, 3, 4, 5, 6])
>>> y2=x.transpose()
>>> y2
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> y3=x.reshape((6,1))
>>> y3
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```

In Numpy, those produced arrays share the same data as the original array in computer memory. In fact they only provide a **view** of the original data by providing different descriptive information such as a new `shape`.

We can see that when the data of `x` is changed, so does all arrays that depend on the same piece of data.

```
>>> x[0,0]=100
>>> y1
array([100,   2,   3,   4,   5,   6])
>>> y2
array([[100,   4],
       [  2,   5],
       [  3,   6]])
>>> y3
array([[100],
       [  2],
       [  3],
       [  4],
       [  5],
       [  6]])
```

As a rule of thumb, in Numpy the data of an array rarely get deep copied. If you want a deep copy, you have to do this explicitly, e.g. by using `ndarray` class' `copy()` method.

Mathematical operators

The `ndarray` class has also defined its own version of operator `+` `-` `*` and `/`. Basically, these are usual mathematical addition subtraction multiplication and division that operates **component-wise** on array elements.

```
>>> x=np.array([[1,2,3], [4,5,6]])
>>> y=x
>>> x+y
array([[ 2,  4,  6],
       [ 8, 10, 12]])
>>> x-y
array([[0, 0, 0],
       [0, 0, 0]])
>>> x*y
array([[ 1,  4,  9],
       [16, 25, 36]])
>>> x/y
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Broadcasting

If the two arrays in a mathematical operations have incompatible shapes, the smaller array is "broadcast" across the larger array so that their shapes become compatible.

- Matrix and scalar

```
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x+1
array([[2, 3, 4],
       [5, 6, 7]])
>>> x*2
array([[ 2,  4,  6],
       [ 8, 10, 12]])
```

Broadcasting

If the two arrays in a mathematical operations have incompatible shapes, the smaller array is "broadcast" across the larger array so that their shapes become compatible.

- Matrix and scalar

```
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x+1
array([[2, 3, 4],
       [5, 6, 7]])
>>> x*2
array([[ 2,  4,  6],
       [ 8, 10, 12]])
```

- Matrix and vector

```
>>> y = np.array([1,2,3])
>>> x + y  # y is added to each row of x
array([[2, 4, 6],
       [5, 7, 9]])
>>> z = np.array([1,2])
>>> x + z  # z cannot be added to rows of x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (2,3)
     (2,)
>>> x + z.reshape((2, 1))  # works!
array([[2, 3, 4],
       [6, 7, 8]])
```

Exercise 7.1

1. Create the following array without using `np.array()`.

$$\mathbf{A} = \begin{pmatrix} 1 & 4 & 9 & 16 & 25 & 36 \\ 49 & 64 & 81 & 100 & 121 & 144 \\ 169 & 196 & 225 & 256 & 289 & 324 \\ 361 & 400 & 441 & 484 & 529 & 576 \end{pmatrix}.$$

2. Compute the variance of each row of $\mathbf{A}$ by exploiting the broadcasting rule.

Exercise 7.1

1. Create the following array without using `np.array()`.

$$\mathbf{A} = \begin{pmatrix} 1 & 4 & 9 & 16 & 25 & 36 \\ 49 & 64 & 81 & 100 & 121 & 144 \\ 169 & 196 & 225 & 256 & 289 & 324 \\ 361 & 400 & 441 & 484 & 529 & 576 \end{pmatrix}.$$

2. Compute the variance of each row of **A** by exploiting the broadcasting rule.

Solution:

```
>>> t = np.arange(1,25)
>>> t2 = t*t
>>> A = t2.reshape((4, 6))
>>> m = A.mean(axis=1)
>>> m = m.reshape((4, 1))
>>> B = A - m
>>> A2 = B*B
>>> var1 = A2.mean(axis=1)
>>> var1
array([  149.13888889,  1059.13888889,  2809.13888889,  5399.13888889])
```

Universal functions

Numpy provides familiar mathematical functions such as `sin()`, `cos()`, and `exp()`. These functions operate element-wise on an array, producing an array as output.

```
>>> x=np.array([[1,2,3],[4,5,6]])
>>> np.log(x)
array([[ 0.        ,  0.69314718,  1.09861229],
       [ 1.38629436,  1.60943791,  1.79175947]])
```

Universal functions

Numpy provides familiar mathematical functions such as `sin()`, `cos()`, and `exp()`. These functions operate element-wise on an array, producing an array as output.

```
>>> x=np.array([[1,2,3],[4,5,6]])
>>> np.log(x)
array([[ 0.        ,  0.69314718,  1.09861229],
       [ 1.38629436,  1.60943791,  1.79175947]])
```

Note that if you use `math.log()` on a Numpy array, you will get an error. This is because `math.log()` only accepts scalar argument.

```
>>> import math
>>> math.log(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars
```

From the examples above, we see that we would have naming conflict should we use `from numpy import *` and `from math import *`.

Linear algebra: matrix multiplication

To compute the product of two matrix A and B, we can use either `np.dot(A, B)` or `A.dot(B)`. They produce the same result.

```
>>> A=np.arange(6).reshape((3, 2))
>>> B=np.arange(-2, 2, 1).reshape((2,2))
>>> A.dot(B)
array([[ 0,  1],
       [-4,  1],
       [-8,  1]])
>>> np.dot(A, B)
array([[ 0,  1],
       [-4,  1],
       [-8,  1]])
```

Linear algebra: matrix multiplication

To compute the product of two matrix A and B, we can use either `np.dot(A, B)` or `A.dot(B)`. They produce the same result.

```
>>> A=np.arange(6).reshape((3, 2))
>>> B=np.arange(-2, 2, 1).reshape((2,2))
>>> A.dot(B)
array([[ 0,  1],
       [-4,  1],
       [-8,  1]])
>>> np.dot(A, B)
array([[ 0,  1],
       [-4,  1],
       [-8,  1]])
```

Care must be taken when do matrix-vector multiplication.

```
>>> A=np.array([[1,2,3], [4,5,6]])
>>> x=np.array([1, -1, 2])
>>> x   # 1-dimensional array
array([ 1, -1,  2])
>>> np.dot(A, x)  # output is 1-dimensional
array([ 5, 11])
>>> y=x.reshape((3,1))
>>> y  # a 2-dimensional array
array([[ 1],
       [-1],
       [ 2]])
>>> np.dot(A, y)  # output is 2-dimensional
array([[ 5],
       [11]])
```

Linear algebra

Numpy contains a linear algebra module named `linalg` which provides a number of useful functions.

`norm()`	Vector or matrix norm
`inv()`	Inverse of a square matrix
`solve()`	Solve a linear system of equations
`det()`	Determinant of a square matrix
`eig()`	Eigenvalues and vectors of a square matrix

Linear algebra

Numpy contains a linear algebra module named `linalg` which provides a number of useful functions.

`norm()`	Vector or matrix norm
`inv()`	Inverse of a square matrix
`solve()`	Solve a linear system of equations
`det()`	Determinant of a square matrix
`eig()`	Eigenvalues and vectors of a square matrix

Depending on the way you import the module, there is a difference on how to use these functions. Take the example of `norm()`:

- If `import numpy as np`, then use `np.linalg.norm()`;
- If `from numpy import linalg`, then use `linalg.norm()`;
- If `from numpy import *`, then use `linalg.norm()`;
- If `from numpy.linalg import *`, then use `norm()`.

Examples

```
>>> from numpy.linalg import *
>>> A=np.array([[1,2],[3,4]])
>>> x=np.array([5,6])
>>> norm(A)  # return Frobenius norm by default for matrix
5.477225575051661
>>> norm(x)  # return Euclidean norm by default for vector
7.810249675906654
>>> inv(A)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> solve(A,x)  # solve equation Ax=b
array([-4. ,  4.5])
>>> det(A)
-2.0000000000000004
>>> v=eig(A)
>>> v  # v[0] is the array of eigenvalues, v[1] is that of eigenvectors
(array([-0.37228132,  5.37228132]), array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]]))
```

To get more details of these functions, press `ctrl` then left-click on the function name from your PyCharm IDE. By doing so, you will be redirected to the original function definition script.

Performance

Numpy is very fast. As an example, to compare the speed your own matrix multiplication with that of `np.dot()`, let us write a test program:

```
import numpy as np
import time

def mydot(A, B):  # naive matrix multiplication
    C = np.zeros((A.shape[0], B.shape[1]))
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            s = 0
            for k in range(A.shape[1]):
                s += A[i, k] * B[k, j]
                C[i, j] = s
    return C

A = np.random.randn(100, 100)
B = np.random.randn(100, 100)

last_time = time.time()
mydot(A, B)
print(time.time() - last_time)
last_time = time.time()
np.dot(A, B)
print(time.time() - last_time)
```

In my computer, the result is

```
0.6555566787719727
0.00850987434387207
```