Numpy multidimensional arrays

By examples

jean-claude.feltes@education.lu

Creation:

`import numpy as np`
`a = np.array([[ 1,  2,  3,  4],`
`               [ 5,  6,  7,  8],`
`               [ 9, 10, 11, 12]])`	

Dimension and shape:

`print (a.ndim)`
`print (a.shape)`
`print (a.size)`
	`2`
	`(3, 4)      # 3 rows, 4 columns`
	`12          # 12 elements`

First dimension: rows, second dimension: columns

Reshape:

`b = a.reshape(4,3)`
`print(b)`
	`[[ 1  2  3]`
	` [ 4  5  6]`
	` [ 7  8  9]`
	` [10 11 12]]`

This might not be what you wanted

The reference trap:

`c = a`
`c[0,0] = 314`
`print(a)`
`print(c)`
	`[[314   2   3   4]`
	` [  5   6   7   8]`
	` [  9  10  11  12]]`
	`[[314   2   3   4]`
	` [  5   6   7   8]`
	` [  9  10  11  12]]`

Setting one or more elements of the copy c to new values, changes not only the copied array c, but also the original array a.
Take care:
The arrays a and c reference the same array, after c = a.
So if an element of c is changed, the same element of a is also changed.
(This is true for other kinds of lists also, in general for mutable objects)

If you do not want this, use the copy function:

`c = `**`a.copy()`**
`c[0,0] = 512`
`print(a)`
`print(c)`
	`[[314   2   3   4]`
	` [  5   6   7   8]`
	` [  9  10  11  12]]`
	`[[512   2   3   4]`
	` [  5   6   7   8]`
	` [  9  10  11  12]]`

(Remember that a[0,0] was changed in the previous example)

Indexing and slicing

Remember that all indexing starts with 0!
So the 2nd row for example has index 1.

Let's begin with the same matrix a:

`import numpy as np`
`a = np.array([[ 1,  2,  3,  4],`
`               [ 5,  6,  7,  8],`
`               [ 9, 10, 11, 12]])`	

Get the second row:

`b = a[1]` `print(b)`	`[5 6 7 8]`

Get the last row:

`c = a[-1]`	`[ 9 10 11 12]`

Get the element in the 3rd row and 2nd column:

`d = a[2, 1]`	`10`

Indexing goes by [row, column]

By the way, another valid syntax would be d = a[2][1]

Get the 3rd column:

`e = a[:, 2]`	`[ 3  7 11]`

Indexing goes by [row, column] and ":" means: all in this, so all rows of column 2

Get a submatrix consisting of 2nd and 3rd columns:

`f = a[:, 1:3]`	`[[ 2  3]`
	` [ 6  7]`
	` [10 11]]`

This gives all rows of the columns with index 1 and 2 (remember that "1:3" means "from 1 to 3-1=2" !
(as 1:3 includes 1, but excludes 3, one of the weirdnesses of Python)

Floating point arrays

The integer trap:

All the above examples were done with integers. Numpy has looked at the defined array a and found it all integers, so the resulting arrays also were integer arrays.

Even an assignment like

a[0,0] = 3.14

would not change the type of the array, the result would be a cast of 3.14 to the ineger value 3.

Defining a floating point array:

<pre>import numpy as np
a = np.array([[3.14, 2,   3,   4],
               [5,   6,   7,   8],
               [9, 10, 11, 12]])
print(a)</pre>
	<pre>[[3.14 2. 3. 4.]
	 [5. 6. 7. 8.]
	 [9. 10. 11. 12.]]</pre>

Even if only one element is a floating point number, Numpy sets all elements to floating point, as can be seen in the result.

Other Numpy functions

Make an array from lists and / or arrays:

<pre>import numpy as np
l1 = (3.14, 2, 5)
l2 = [2, 3, 4]
l = l1, l2
print(l)
a = **np.asarray(l)**
print(a)</pre>
	<pre>((3.14, 2, 5), [2, 3, 4])
	
	[[3.14 2. 5.]
	 [2. 3. 4.]]</pre>

Make zeros array:

<pre># one dimensional: 	
z = np.zeros(5)
print(z)
 	
# 2 dimensional 	
z2 = np.zeros((2,3))
print(z2)</pre>
	<pre>
	[0. 0. 0. 0. 0.]
	
	[[0. 0. 0.]
	 [0. 0. 0.]]</pre>

Make an array of evenly spaced numbers:

Example: 5 values between 2 and 3

`l = np.linspace(2,3,5)`	`[2.   2.25 2.5  2.75 3.  ]`

Operations on arrays

Operations are done element wise.

```
import numpy as np
a = np.array([[ 1,  2,  3,  4],
              [ 5,  6,  7,  8],
              [ 9, 10, 11, 12]])
b = a * 2
print(b)

c = a + b
print(c)
```

```
[[ 2  4  6  8]      #b
 [10 12 14 16]
 [18 20 22 24]]

[[ 3  6  9 12]      #c
 [15 18 21 24]
 [27 30 33 36]]
```

This is also true for multiplication and division.

Functions can directly operate on arrays:

```
d = np.sin(a)
print(d)
```

```
[[ 0.84147098  0.90929743  0.14112001 -0.7568025 ]
 [-0.95892427 -0.2794155   0.6569866   0.98935825]
 [ 0.41211849 -0.54402111 -0.99999021 -0.53657292]]
```

Calculations with constants are also done elementwise:

```
e = a + 5
print(e)
```

```
[[ 6  7  8  9]
 [10 11 12 13]
 [14 15 16 17]]
```

Mathematical Matrix operations

Matrix multiplication → dot function

```
import numpy as np
a = np.array([[ 1,  2 ],
              [ 3,  4 ]])

b = np.array([[5, 6],
             [7, 8]])

c = np.dot(a,b)
print(c)
```

```
[[19 22]
 [43 50]]
```