NumPy Tutorial

The Basics

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In Numpy dimensions are called axes. The number of axes is rank.

For example, the coordinates of a point in 3D space \([1, 2, 1]\) is an array of rank 1, because it has one axis. That axis has a length of 3. In example pictured below, the array has rank 2 (it is 2-dimensional). The first dimension (axis) has a length of 2, the second dimension has a length of 3.

```
In [1]: [[ 1.,  0.,  0.],
          [ 0.,  1.,  2.]]
Out[1]: [[1.0, 0.0, 0.0], [0.0, 1.0, 2.0]]
```
Numpy's array class is called ndarray. It is also known by the alias array. Note that numpy.array is not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays and offers less functionality. The more important attributes of an ndarray object are:

**ndarray.ndim**
the number of axes (dimensions) of the array. In the Python world, the number of dimensions is referred to as rank.

**ndarray.shape**
the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the rank, or number of dimensions, ndim.

**ndarray.size**
the total number of elements of the array. This is equal to the product of the elements of shape.

**ndarray.dtype**
an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

An example:

```
In [2]: from numpy import *
   ...: a = arange(15).reshape(3, 5)
   ...: a
Out[2]: array([[ 0,  1,  2,  3,  4],
               [ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14]])

In [3]: a.shape
Out[3]: (3, 5)

In [4]: a.ndim
Out[4]: 2

In [5]: a.dtype.name
Out[5]: 'int64'
```
Array Creation

There are several ways to create arrays. For example, you can create an array from a regular Python list or tuple using the array function. The type of the resulting array is deduced from the type of the elements in the sequences.

```python
In [6]: a.size
Out[6]: 15

In [7]: type(a)
Out[7]: numpy.ndarray

In [8]: b = array([6, 7, 8])
   ...
Out[8]: array([6, 7, 8])

In [9]: type(b)
Out[9]: numpy.ndarray

In [10]: from numpy import *
   ...
a = array([2, 3, 4])
   ...
a
Out[10]: array([2, 3, 4])

In [11]: a.dtype
Out[11]: dtype('int64')

In [12]: b = array([1.2, 3.5, 5.1])

In [13]: b.dtype
Out[13]: dtype('float64')
```

A frequent error consists in calling array with multiple numeric arguments, rather than providing a single list of numbers as an argument.
In [14]:
   a = array(1, 2, 3, 4)  # WRONG

---

ValueErrorTraceback (most recent call last)
<ipython-input-14-31acf3eb6ef4> in <module>()
----> 1 a = array(1, 2, 3, 4)  # WRONG

ValueError: only 2 non-keyword arguments accepted

In [16]:
   a = array([1, 2, 3, 4])  # RIGHT

array transforms sequences of sequences into two-dimensional arrays, sequences of
sequences of sequences into three-dimensional arrays, and so on.

In [17]:
   b = array([[1.5, 2, 3],
               [4, 5, 6]])

Out[17]:
   array([[ 1.5,  2. ,  3. ],
           [ 4. ,  5. ,  6. ]])

The type of the array can also be explicitly specified at creation time:

In [18]:
   c = array([[1,2], [3,4]], dtype=complex)

Out[18]:
   array([[ 1.+0.j,  2.+0.j],
           [ 3.+0.j,  4.+0.j]])

Often, the elements of an array are originally unknown, but its size is known. Hence,
NumPy offers several functions to create arrays with initial placeholder content. These
minimize the necessity of growing arrays, an expensive operation.

The function zeros creates an array full of zeros, the function ones creates an array full
of ones, and the function empty creates an array whose initial content is random and
depends on the state of the memory. By default, the dtype of the created array is
float64.
To create sequences of numbers, NumPy provides a function analogous to range that returns arrays instead of lists.

```python
In [22]: arange(10, 30, 5)
Out[22]: array([10, 15, 20, 25])
```

When `arange` is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the function `linspace` that receives as an argument the number of elements that we want, instead of the step:

```python
In [24]: linspace(0, 2, 9)  # 9 numbers from 0 to 2
Out[24]: array([ 0.    , 0.25  , 0.5   , 0.75 , 1.    , 1.25 , 1.5   , 1.75 , 2.    ])
```

```python
In [25]: x = linspace(0, 2*pi, 100)  # useful to evaluate function at lots of points
f = sin(x)
```
See also:
array, zeros, zeros_like, ones, ones_like, empty, empty_like, arange, linspace, rand, randn, fromfunction, fromfile

**Printing Arrays**

When you print an array, NumPy displays it in a similar way to nested lists, but with the following layout:

- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

One-dimensional arrays are then printed as rows, bidimensionals as matrices and tridimensionals as lists of matrices.

```python
In [26]: a = arange(6) # 1d array
a
```

```
Out[26]: array([0, 1, 2, 3, 4, 5])
```

```python
In [27]: b = arange(12).reshape(4, 3) # 2d array
b
```

```
Out[27]: array([[0, 1, 2],
              [3, 4, 5],
              [6, 7, 8],
              [9, 10, 11]])
```

```python
In [28]: c = arange(24).reshape(2, 3, 4) # 3d array
c
```

```
Out[28]: array([[[0, 1, 2, 3],
               [4, 5, 6, 7],
               [8, 9, 10, 11]],
               [[12, 13, 14, 15],
               [16, 17, 18, 19],
               [20, 21, 22, 23]]])
```
If an array is too large to be printed, NumPy automatically skips the central part of the array and only prints the corners:

```python
In [29]: arange(10000)
Out[29]: array([ 0,  1,  2, ..., 997, 998, 999])
```

```python
In [30]: arange(10000).reshape(100, 100)
Out[30]: array([[  0,  1,  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]])
```

To disable this behaviour and force NumPy to print the entire array, you can change the printing options using `set_printoptions`.

```python
def set_printoptions(threshold='nan')
```

**Basic Operations**

Arithmetic operators on arrays apply elementwise. A new array is created and filled with the result.

```python
In [31]: a = array([[20, 30, 40, 50]])
b = arange(4)
b
Out[31]: array([0, 1, 2, 3])
```

```python
In [32]: c = a - b
c
Out[32]: array([20, 29, 38, 47])
```

```python
In [33]: b**2
Out[33]: array([0, 1, 4, 9])
```
Unlike in many matrix languages, the product operator * operates elementwise in NumPy arrays. The matrix product can be performed using the dot function or creating matrix objects.

Some operations, such as += and *=, act in place to modify an existing array rather than create a new one.
When operating with arrays of different types, the type of the resulting array corresponds to the more general or precise one (a behavior known as upcasting).

```python
In [41]: a = ones(3, dtype=int32)
b = linspace(0, pi, 3)
b.dtype.name
Out[41]: 'float64'
```

```python
In [42]: c = a + b
c
Out[42]: array([ 1. ,  2.57079633,  4.14159265])
```

```python
In [43]: c.dtype.name
Out[43]: 'float64'
```

```python
In [44]: d = exp(c * 1j)
d
Out[44]: array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j, -0.54030231-0.84147098j])
```

```python
In [45]: d.dtype.name
Out[45]: 'complex128'
```

Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the ndarray class.

```python
In [46]: a = random.random((2, 3))
a
Out[46]: array([[ 0.62602758,  0.46900816,  0.08922625],
               [ 0.08886244,  0.86957791,  0.58682786]])
```
By default, these operations apply to the array as though it were a list of numbers, regardless of its shape. However, by specifying the axis parameter you can apply an operation along the specified axis of an array:

```
In [48]: b = arange(12).reshape(3, 4)
b
Out[48]: array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]])
```

```
In [49]: b.sum(axis=0)  # sum of each column
Out[49]: array([12, 15, 18, 21])
```

```
In [50]: b.min(axis=1)  # min of each row
Out[50]: array([0, 4, 8])
```

```
In [51]: b.cumsum(axis=1)  # cumulative sum along each row
Out[51]: array([[ 0,  1,  3,  6],
               [ 4,  9, 15, 22],
               [ 8, 17, 27, 38]])
```

**Universal Functions**

NumPy provides familiar mathematical functions such as sin, cos, and exp. In NumPy, these are called "universal functions"(ufunc). Within NumPy, these functions operate elementwise on an array, producing an array as output.

```
In [52]: B = arange(3)
B
Out[52]: array([0, 1, 2])
```
Indexing, Slicing and Iterating

One-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences.

```
In [55]: a = arange(10)**3
    
a
Out[55]: array([  0,    1,    8,   27,   64,  125,  216,  343,  512,  729])

In [56]: a[2]
    
Out[56]: 8

In [57]: a[2:5]
    
Out[57]: array([  8,   27,   64])

In [58]: a[6:2] = -1000  # equivalent to a[0:6:2] = -1000; from start to position 6, exclusive, set every 2nd element to -1000
    
a
Out[58]: array([-1000,   1, -1000,   27, -1000,  125,  216,  343,  512,  729])
```
Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas:

```
In [60]: def f(x,y):
   ...:     return 10*x + y
   ...:
   ...:     b = fromfunction(f, (5, 4), dtype=int)
   ...:     b

Out[60]: array([[  0,   1,   2,   3],
                 [ 10,  11,  12,  13],
                 [ 20,  21,  22,  23],
                 [ 30,  31,  32,  33],
                 [ 40,  41,  42,  43]])
```

```
In [61]: b[2, 3]

Out[61]: 23
```

```
In [62]: b[0:5, 1]  # each row in the second column of b

Out[62]: array([ 1, 11, 21, 31, 41])
```

```
In [63]: b[:, 1]  # equivalent to the previous example

Out[63]: array([ 1, 11, 21, 31, 41])
```

```
In [64]: b[1:3, :]  # each column in the second and third row of b

Out[64]: array([[10, 11, 12, 13],
                 [20, 21, 22, 23]])
```

When fewer indices are provided than the number of axes, the missing indices are considered complete slices:

```
In [65]: b[-1]  # the last row. Equivalent to b[-1, :]

Out[65]: array([40, 41, 42, 43])
```
The expression within brackets in \( b[i] \) is treated as an \( i \) followed by as many instances of \( : \) as needed to represent the remaining axes. NumPy also allows you to write this using dots as \( b[i, \ldots] \).

The dots (...) represent as many colons as needed to produce a complete indexing tuple. For example, if \( x \) is a rank 5 array (i.e., it has 5 axes), then

\[
x[1,2,\ldots] \text{ is equivalent to } x[1,2,:,\ldots], \]
\[
x[\ldots,3] \text{ to } x[\ldots,:,3] \text{ and} \]
\[
x[4,\ldots,5,:] \text{ to } x[4,:,5,:].
\]

```
In [66]: c = array([[[ 0,  1,  2],
                  [ 10, 12, 13]],
                 [[100,101,102],
                  [110,112,113]])

In [66]: c
Out[66]: (2, 2, 3)
```

```
In [67]: c[1,...] # same as c[1,:,] or c[1]
Out[67]: array([[100, 101, 102],
               [110, 112, 113]])
```

```
In [68]: c[...,:2] # same as c[::,2]
Out[68]: array([[   2,   13],
               [102, 113]])
```

Iterating over multidimensional arrays is done with respect to the first axis:

```
In [69]: for row in b:
   print(row)
   [0 1 2 3]
   [10 11 12 13]
   [20 21 22 23]
   [30 31 32 33]
   [40 41 42 43]
```
However, if one wants to perform an operation on each element in the array, one can use the flat attribute which is an iterator over all the elements of the array:

```python
In [70]: for element in b.flat:
    print(element, end=','
0,1,2,3,10,11,12,13,20,21,22,23,30,31,32,33,40,41,42,43,
```

See also

[], ..., newaxis, ndenumerate, indices, index exp

**MATLAB**

For MATLAB users there is a [cheatsheet](#) available for common commands.

```python
In [71]: from IPython.core.display import HTML
def css_styling():
    styles = open("custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Out[71]:

---

[cheatsheet]: # (https://example.com/cheatsheet)