```
In [1]:   # My standard magic !  You will see this in almost all my notebooks.

          from IPython.core.interactiveshell import InteractiveShell
          InteractiveShell.ast_node_interactivity = "all"

          # Reload all modules imported with %aimport
          %load_ext autoreload
          %autoreload 1

          %matplotlib inline
```

NumPy

[VandePlas Chapter 2 (external/PythonDataScienceHandbook/notebooks/02.00-Introduction-to-NumPy.ipynb)](), [Geron notebook (external/handson-ml/tools_numpy.ipynb)]()

Python lists

Lists are *heterogeneous*: can contain elements of mixed type

```
In [2]:  l = list( range(0,10) )
         print(l)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [3]:  l[2] = "two"
         print(l)
```

```
[0, 1, 'two', 3, 4, 5, 6, 7, 8, 9]
```

Heterogeneity == *slow*

- Python interpreter has to constantly examine types

NumPy ndarray

```
In [4]:  import numpy as np
```

NumPy n-dimensional arrays (ndarray) are *homogenous*

- Can be faster because don't waste time examining type of each element
- Can be treated as vectors
- Vector arithmetic via compiled code = *fast*

```
In [5]:   l = list( range(0,10))
          l_plus_1 = [ e+1 for e in l]
          print(l_plus_1)
```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```
In [6]:   l_np = np.array( np.arange(0,10))
          print(l_np +1)
```

[1 2 3 4 5 6 7 8 9 10]

Speed comparison

```
In [7]:   list_len = 1000
          l = list( range(0, list_len))
          %timeit [ e+1 for e in l]
```

53.8 µs ± 780 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

```
In [8]:   l_np = np.array( np.arange(0, list_len) )
          %timeit l_np +1
```

2.6 µs ± 136 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

When dealing with large datasets, you need NumPy

Basics of NumPy arrays

[VanderPlas (external/PythonDataScienceHandbook/notebooks/02.02-The-Basics-Of-NumPy-Arrays.ipynb)](external/PythonDataScienceHandbook/notebooks/02.02-The-Basics-Of-NumPy-Arrays.ipynb)

[Vandeplas YouTube: Losing your loops - slides (https://speakerdeck.com/jakevdp/losing-your-loops-fast-numerical-computing-with-numpy-pycon-2015?slide=14)](https://speakerdeck.com/jakevdp/losing-your-loops-fast-numerical-computing-with-numpy-pycon-2015?slide=14)

The most operation on ndarrays is indexing.

- ndarray indices are 0-based (i.e, first row/col is numbered 0, not 1)

```
In [9]:  x = np.arange(0,6)

         x

         x[2]

         M = np.arange(0,6).reshape(2,3)
         M

         M[1,1]
```

Out[9]: `array([0, 1, 2, 3, 4, 5])`

Out[9]: 2

Out[9]: `array([[0, 1, 2],`
         `       [3, 4, 5]])`

Out[9]: 4

Slicing

- Python (not just NumPy) upper bound of index is NOT inclusive

```
In [10]:  print("x: ", x)
          print("x tail: ", x[2:])
          print("x head: ", x[:2])
```

```
x:  [0 1 2 3 4 5]
x tail:  [2 3 4 5]
x head:  [0 1]
```

Strides

```
x[start:stop:step]
```

In [11]: `x[1:5:2]`

Out[11]: `array([1, 3])`

Reshaping

In [12]:
```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Add dimensions

```
In [13]:  x = np.arange(0,6)
          print("x: ", x)
          print("x shape: ", x.shape)

          print("x re-shaped: ", x.reshape(1,-1))
          print("x re-shaped shape: ", x.reshape(1,-1).shape)

          print("x w/newaxis: ", x[ np.newaxis,:])
          print("x w/newaxis sja[e: ", x[ np.newaxis,:].shape)
```

```
x:  [0 1 2 3 4 5]
x shape:  (6,)
x re-shaped:  [[0 1 2 3 4 5]]
x re-shaped shape:  (1, 6)
x w/newaxis:  [[0 1 2 3 4 5]]
x w/newaxis sja[e:  (1, 6)
```

Concatentation, splitting

```
In [14]:  x = np.array([1, 2, 3])
          y = np.array([3, 2, 1])
          x
          y

          np.concatenate([x, y])
```

Out[14]: `array([1, 2, 3])`

Out[14]: `array([3, 2, 1])`

Out[14]: `array([1, 2, 3, 3, 2, 1])`

You can concatenate multi-dimensional ndarrays:

```
In [15]:  M1 = np.array([ [1, 2, 3],
                          [4, 5, 6]
                        ])

          M2 = np.array([ [ 7,  8,  9 ],
                          [ 10, 11, 12]
                        ])

          M1
          M2

          np.concatenate([ M1, M2 ])
```

```
Out[15]:  array([[1, 2, 3],
                 [4, 5, 6]])
```

```
Out[15]:  array([[ 7,  8,  9],
                 [10, 11, 12]])
```

```
Out[15]:  array([[ 1,  2,  3],
                 [ 4,  5,  6],
                 [ 7,  8,  9],
                 [10, 11, 12]])
```

You can also specify the dimension on which to concatenate

```
M1
M2

np.concatenate([ M1, M2 ], axis=1)
```

Out[16]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

Out[16]:
```
array([[ 7,  8,  9],
       [10, 11, 12]])
```

Out[16]:
```
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

You can also use `vstack` (vertical stack) and `hstack` (horizontal stack)

```
In [17]:  x = np.array([1, 2, 3])
          grid = np.array([[9, 8, 7],
                           [6, 5, 4]])

          y = np.array( [ [100],
                          [200]
                        ])
          x
          grid

          print("vstack:")
          # vertically stack the arrays
          np.vstack([x, grid])

          print("hstack:")
          y
          grid
          np.hstack( [y, grid])
```

```
Out[17]:  array([1, 2, 3])
```

```
Out[17]:  array([[9, 8, 7],
                 [6, 5, 4]])
```

```
          vstack:
```

```
Out[17]:  array([[1, 2, 3],
                 [9, 8, 7],
                 [6, 5, 4]])
```

```
          hstack:
```

```
Out[17]:  array([[100],
                 [200]])
```

```
Out[17]:  array([[9, 8, 7],
                 [6, 5, 4]])
```

```
Out[17]:  array([[100,    9,    8,    7],
                 [200,    6,    5,    4]])
```

Ufuncs

[Vandeplass (external/PythonDataScienceHandbook/notebooks/02.03-Computation-on-arrays-ufuncs.ipynb#Introducing-UFuncs)](external/PythonDataScienceHandbook/notebooks/02.03-Computation-on-arrays-ufuncs.ipynb#Introducing-UFuncs)

Math

- element-wise operations
- vectorized for speed

- operator overloading

  - +, -, *, /

  - <, ==, >

  - provides natural syntax

    - `l + 1`
    - `` `np.add(l,1) ``

```
In [18]:  x = np.array( np.arange(0,10))
          print("x: ", x)
          print("+1: ", x + 1)
          print("+1 verbose: ", np.add(x,1))
          print("-1: ", x -1)
```

```
x:  [0 1 2 3 4 5 6 7 8 9]
+1:  [ 1  2  3  4  5  6  7  8  9 10]
+1 verbose:  [ 1  2  3  4  5  6  7  8  9 10]
-1:  [-1  0  1  2  3  4  5  6  7  8]
```

Aggregates

- Aggregation: taking a one-dimensional slice of an ndarray and reducing it to a scalar
    - also known as **reduce**

Best illustrated with an example

```
In [19]:  x = np.arange(1, 6)
          print("x: ", x)
          print("x reduced by add: ",np.add.reduce(x))

          # Less verbose synonym
          print("x reduced by add, via sum", x.sum())
```

```
x:  [1 2 3 4 5]
x reduced by add:   15
x reduced by add, via sum 15
```

Aggregates on multi-dimensional ndarray: choose your dimension

```
In [20]:  x = np.arange(1,7).reshape(2,3)
          print("x: ", x)

          print("x reduced on first dimension: ", x.sum(axis=0))

          print("x reduced on second dimension: ", x.sum(axis=1))
```

```
x:  [[1 2 3]
 [4 5 6]]
x reduced on first dimension:  [5 7 9]
x reduced on second dimension:  [ 6 15]
```

Cumulative

Closely related to reduce : accumulate

- running operations, e.g, running sum

```
print("x: ", x)
print("x running sum: ", np.add.accumulate(x))   # NOTE: not a method ON x; x is
 a parameter

# Less verbose synonym
print("x running sum via cumsum: ", x.cumsum())
```

```
x:  [[1 2 3]
 [4 5 6]]
x running sum:  [[1 2 3]
 [5 7 9]]
x running sum via cumsum:  [ 1  3  6 10 15 21]
```

Broadcasting

[Vanderplass (external/PythonDataScienceHandbook/notebooks/02.05-Computation-on-arrays-broadcasting.ipynb)](external/PythonDataScienceHandbook/notebooks/02.05-Computation-on-arrays-broadcasting.ipynb)

You hopefully intuitively understand what NumPy does when a binary operator is applied to 2 identically-shaped arguments

In [22]:
```python
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

Out[22]: `array([5, 6, 7])`

But what happens if the two arguments have different shape ? Simplest case: one argument is dimension 0 or 1:

In [23]:
```python
print("a: ", a)
print("a + 1: ", a+1)
```

```
a:  [0 1 2]
a + 1:  [1 2 3]
```

Next case: what if one argument is identical to the other EXCEPT is missing a dimension:

```python
M = np.arange(1,10).reshape(3,3)

print("a shape (", a.shape, "): ", a)
print("M shape (", M.shape, "):\n", M)
print("a + M shape(", (a+M).shape, "):\n", a + M)
```

```
a shape ( (3,) ):  [0 1 2]
M shape ( (3, 3) ):
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
a + M shape( (3, 3) ):
 [[ 1  3  5]
 [ 4  6  8]
 [ 7  9 11]]
```

NumPy took a one dimensional ndarray a and treated it like a 2-d ndarray by repeated it's rows

This is called **broadcasting**

Broadcasting follows some simple rules (quoted from Vanderplass (external/PythonDataScienceHandbook/notebooks/02.05-Computation-on-arrays-broadcasting.ipynb)):

Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.
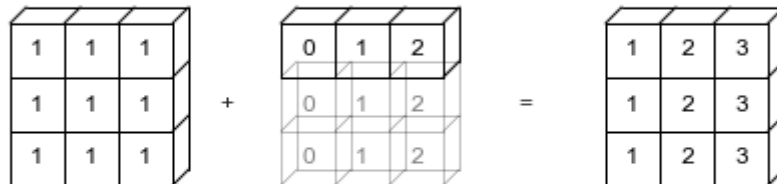
Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.

Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

np. arange(3)+5

| 0 | 1 | 2 |

+

| 5 | 5 | 5 |

=

| 5 | 6 | 7 |

np. ones((3, 3))+np. arange(3)

1	1	1
1	1	1
1	1	1

+

0	1	2
0	1	2
0	1	2

=

1	2	3
1	2	3
1	2	3

np. arange(3).reshape((3, 1))+np. arange(3)

0	0	0
1	1	1
2	2	2

+

0	1	2
0	1	2
0	1	2

=

0	1	2
1	2	3
2	3	4