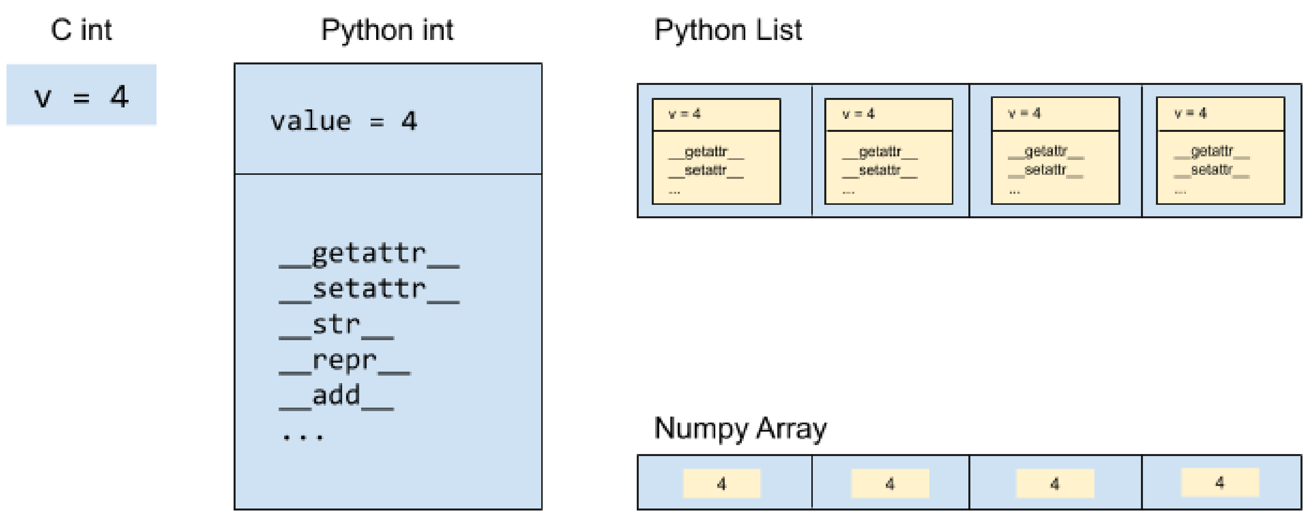
Numpy: Numeric computing library

NumPy (Numerical Python) is one of the core packages for numerical computing in Python. Pandas, Matplotlib, Statmodels and many other Scientific libraries rely on NumPy.

NumPy major contributions are:

- Efficient numeric computation with C primitives
- Efficient collections with vectorized operations
- An integrated and natural Linear Algebra API
- A C API for connecting NumPy with libraries written in C, C++, or FORTRAN.

Let's develop on efficiency. In Python, **everything is an object**, which means that even simple ints are also objects, with all the required machinery to make object work. We call them "Boxed Ints". In contrast, NumPy uses primitive numeric types (floats, ints) which makes storing and computation efficient.



Hands on!

In [4]:

```
import sys
import numpy as np
```

Basic Numpy Arrays

In [5]:

```
np.array([1, 2, 3, 4])
```

Out[5]:

```
array([1, 2, 3, 4])
```

In [6]:

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

In [7]:

```
b = np.array([0, .5, 1, 1.5, 2])
```

In [8]:

```
a[0], a[1]
```

Out[8]:

```
(1, 2)
```

In [9]:

```
a[0:]
```

Out[9]:

```
array([1, 2, 3, 4])
```

In [10]:

```
a[1:3]
```

Out[10]:

```
array([2, 3])
```

In [11]:

```
a[1:-1]
```

Out[11]:

```
array([2, 3])
```

In [12]:

```
a[::2]
```

Out[12]:

```
array([1, 3])
```

In [13]:

```
b
```

Out[13]:

```
array([0. , 0.5, 1. , 1.5, 2. ])
```

In [14]:

```
b[0], b[2], b[-1]
```

Out[14]:

```
(0.0, 1.0, 2.0)
```

In [15]:

```
b[[0, 2, -1]]
```

Out[15]:

```
array([0., 1., 2.])
```

Array Types

In [16]:

```
a
```

Out[16]:

```
array([1, 2, 3, 4])
```

In [17]:

```
a.dtype
```

Out[17]:

```
dtype('int64')
```

In [18]:

```
b
```

Out[18]:

```
array([0. , 0.5, 1. , 1.5, 2. ])
```

In [19]:

```
b.dtype
```

Out[19]:

```
dtype('float64')
```

In [20]:

```
np.array([1, 2, 3, 4], dtype=np.float)
```

Out[20]:

```
array([1., 2., 3., 4.])
```

In [21]:

```
np.array([1, 2, 3, 4], dtype=np.int8)
```

Out[21]:

```
array([1, 2, 3, 4], dtype=int8)
```

In [22]:

```
c = np.array(['a', 'b', 'c'])
```

In [23]:

```
c.dtype
```

Out[23]:

```
dtype('<U1')
```

In [24]:

```
d = np.array([{'a': 1}, sys])
```

In [25]:

```
d.dtype
```

Out[25]:

```
dtype('O')
```

Dimensions and shapes

In [26]:

```
A = np.array([
    [1, 2, 3],
    [4, 5, 6]
])
```

In [27]:

```
A.shape
```

Out[27]:

```
(2, 3)
```

In [28]:

```
A.ndim # How many dimensions it has
```

Out[28]:

```
2
```

In [29]:

```
A.size # Total number of elements
```

Out[29]:

```
6
```

In [27]:

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

In [28]:

```
B
```

Out[28]:

```
array([[[12, 11, 10],
        [ 9,  8,  7]],

       [[ 6,  5,  4],
        [ 3,  2,  1]]])
```

In [29]:

```
B.shape # multidimensional
```

Out[29]:

```
(2, 2, 3)
```

In [30]:

```
B.ndim
```

Out[30]:

```
3
```

In [31]:

```
B.size
```

Out[31]:

```
12
```

If the shape isn't consistent, it'll just fall back to regular Python objects:

In [32]:

```
C = np.array([
    [
        [12, 11, 10],
        [9, 8, 7],
    ],
    [
        [6, 5, 4]
    ]
])
```

In [33]:

```
C.dtype
```

Out[33]:

```
dtype('O')
```

In [34]:

```
C.shape
```

Out[34]:

```
(2,)
```

In [35]:

```
C.size
```

Out[35]:

```
2
```

In [ ]:

```
type(C[0])
```

Indexing and Slicing of Matrices

In [36]:

```
# Square matrix
A = np.array([
#.    0. 1. 2
     [1, 2, 3], # 0
     [4, 5, 6], # 1
     [7, 8, 9]  # 2
])
```

In [37]:

```
A[1]
```

Out[37]:

```
array([4, 5, 6])
```

In [38]:

```
A[1][0]
```

Out[38]:

```
4
```

In [ ]:

```
# A[d1, d2, d3, d4]
```

In [39]:

```
A[1, 0]
```

Out[39]:

```
4
```

In [40]:

```
A[0:2]
```

Out[40]:

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

In [41]:

```
A[:, :2]
```

Out[41]:

```
array([[1, 2],
       [4, 5],
       [7, 8]])
```

In [42]:

```
A[:2, :2]
```

Out[42]:

```
array([[1, 2],
       [4, 5]])
```

In [43]:

```
A[:2, 2:]
```

Out[43]:

```
array([[3],
       [6]])
```

In [44]:

```
A
```

Out[44]:

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

In [45]:

```
A[1] = np.array([10, 10, 10])
```

In [46]:

```
A
```

Out[46]:

```
array([[ 1,  2,  3],
       [10, 10, 10],
       [ 7,  8,  9]])
```

In [47]:

```
A[2] = 99
```

In [48]:

```
A
```

Out[48]:

```
array([[ 1,  2,  3],
       [10, 10, 10],
       [99, 99, 99]])
```

Summary statistics

In [49]:

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

In [50]:

```
a.sum()
```

Out[50]:

```
10
```

In [51]:

```
a.mean()
```

Out[51]:

```
2.5
```

In [52]:

```
a.std()
```

Out[52]:

```
1.118033988749895
```

In [53]:

```
a.var()
```

Out[53]:

```
1.25
```

In [54]:

```
A = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])
```

In [55]:

```
A.sum()
```

Out[55]:

```
45
```

In [56]:

```
A.mean()
```

Out[56]:

```
5.0
```

In [57]:

```
A.std()
```

Out[57]:

```
2.581988897471611
```

In [58]:

```
A.sum(axis=0) # Columns
```

Out[58]:

```
array([12, 15, 18])
```

In [59]:

```
A.sum(axis=1) # Rows
```

Out[59]:

```
array([ 6, 15, 24])
```

In [60]:

```
A.mean(axis=0)
```

Out[60]:

```
array([4., 5., 6.])
```

In [61]:

```
A.mean(axis=1)
```

Out[61]:

```
array([2., 5., 8.])
```

In [62]:

```
A.std(axis=0)
```

Out[62]:

```
array([2.44948974, 2.44948974, 2.44948974])
```

In [63]:

```
A.std(axis=1)
```

Out[63]:

```
array([0.81649658, 0.81649658, 0.81649658])
```

And [many more (https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.ndarray.html#array-methods)](https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.ndarray.html#array-methods)...

Broadcasting and Vectorized operations

In [66]:

```
a = np.arange(4)
```

In [67]:

```
a
```

Out[67]:

```
array([0, 1, 2, 3])
```

In [68]:

```
a + 10 # adds 10 to every element within the array
```

Out[68]:

```
array([10, 11, 12, 13])
```

In [69]:

```
a * 10 #multiplies every element in the array by 10
```

Out[69]:

```
array([ 0, 10, 20, 30])
```

In [70]:

```
a
```

Out[70]:

```
array([0, 1, 2, 3])
```

In [71]:

```
a += 100 # Add 100 to every element in the array
```

In [73]:

```
a
```

Out[73]:

```
array([100, 101, 102, 103])
```

In [74]:

```
l = [0, 1, 2, 3]
```

In [75]:

```
[i * 10 for i in l] # list comprehension
```

Out[75]:

```
[0, 10, 20, 30]
```

In [76]:

```
a = np.arange(4)
```

In [77]:

```
a
```

Out[77]:

```
array([0, 1, 2, 3])
```

In [78]:

```
b = np.array([10, 10, 10, 10])
```

In [79]:

```
b
```

Out[79]:

```
array([10, 10, 10, 10])
```

In [80]:

```
a + b  # you can add arrays
```

Out[80]:

```
array([10, 11, 12, 13])
```

In [81]:

```
a * b # you can multiply arrays
```

Out[81]:

```
array([ 0, 10, 20, 30])
```

Boolean arrays

(Also called masks)

In [82]:

```
a = np.arange(4)
```

In [83]:

```
a
```

Out[83]:

```
array([0, 1, 2, 3])
```

In [85]:

```
a[0], a[-1]
```

Out[85]:

```
(0, 3)
```

In [84]:

```
a[[0, -1]]
```

Out[84]:

```
array([0, 3])
```

In [86]:

```
a[[True, False, False, True]]
```

Out[86]:

```
array([0, 3])
```

In [89]:

```
a
```

Out[89]:

```
array([0, 1, 2, 3])
```

In [88]:

```
a >= 2
```

Out[88]:

```
array([False, False,  True,  True])
```

In [90]:

```
a[a >= 2]
```

Out[90]:

```
array([2, 3])
```

In [91]:

```
a.mean()
```

Out[91]:

```
1.5
```

In [92]:

```
a[a > a.mean()]
```

Out[92]:

```
array([2, 3])
```

In [93]:

```
a[~(a > a.mean())] # not greater than the mean
```

Out[93]:

```
array([0, 1])
```

In [94]:

```
a[(a == 0) | (a == 1)] # or / and in python
```

Out[94]:

```
array([0, 1])
```

In [95]:

```
a[(a <= 2) & (a % 2 == 0)]
```

Out[95]:

```
array([0, 2])
```

In [96]:

```
A = np.random.randint(100, size=(3, 3))
```

In [97]:

```
A
```

Out[97]:

```
array([[71,  6, 42],
       [40, 94, 24],
       [ 2, 85, 36]])
```

In [98]:

```
A[np.array([
    [True, False, True],
    [False, True, False],
    [True, False, True]
])]
```

Out[98]:

```
array([71, 42, 94,  2, 36])
```

In [99]:

```
A > 30
```

Out[99]:

```
array([[ True, False,  True],
       [ True,  True, False],
       [False,  True,  True]])
```

In [100]:

```
A[A > 30]
```

Out[100]:

```
array([71, 42, 40, 94, 85, 36])
```

Linear Algebra

In [101]:

```
A = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])
```

In [102]:

```
B = np.array([
    [6, 5],
    [4, 3],
    [2, 1]
])
```

In [103]:

```
A.dot(B)
```

Out[103]:

```
array([[20, 14],
       [56, 41],
       [92, 68]])
```

In [104]:

```
A @ B # cross products
```

Out[104]:

```
array([[20, 14],
       [56, 41],
       [92, 68]])
```

In [105]:

```
B.T # transposing matrices
```

Out[105]:

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

In [106]:

```
A
```

Out[106]:

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

In [107]:

```
B.T @ A
```

Out[107]:

```
array([[36, 48, 60],
       [24, 33, 42]])
```

Size of objects in Memory

Int, floats

In [108]:

```
# An integer in Python is > 24bytes
sys.getsizeof(1)
```

Out[108]:

```
28
```

In [109]:

```
# Longs are even larger
sys.getsizeof(10**100)
```

Out[109]:

```
72
```

In [110]:

```
# Numpy size is much smaller
np.dtype(int).itemsize
```

Out[110]:

```
8
```

In [112]:

```
# Numpy size is much smaller
np.dtype(np.int8).itemsize
```

Out[112]:

```
1
```

In [111]:

```
np.dtype(float).itemsize
```

Out[111]:

```
8
```

Lists are even larger

In [ ]:

```
# A one-element list
sys.getsizeof([1])
```

In [ ]:

```
# An array of one element in numpy
np.array([1]).nbytes
```

And performance is also important

In [117]:

```
l = list(range(100000))
```

In [118]:

```
a = np.arange(100000)
```

In [119]:

```
%time np.sum(a ** 2) # timing the python commmands
```

```
CPU times: user 1.06 ms, sys: 279 µs, total: 1.34 ms
Wall time: 701 µs
```

Out[119]:

```
333328333350000
```

In [120]:

```
%time sum([x ** 2 for x in l])
```

```
CPU times: user 36.1 ms, sys: 0 ns, total: 36.1 ms
Wall time: 35.5 ms
```

Out[120]:

```
333328333350000
```

Useful Numpy functions

`random`

In [ ]:

```
np.random.random(size=2)
```

In [ ]:

```
np.random.normal(size=2)
```

In [ ]:

```
np.random.rand(2, 4)
```

`arange`

In [ ]:

```
np.arange(10)
```

In [ ]:

```
np.arange(5, 10)
```

In [ ]:

```
np.arange(0, 1, .1)
```

`reshape`

In [ ]:

```
np.arange(10).reshape(2, 5)
```

In [ ]:

```
np.arange(10).reshape(5, 2)
```

---

linspace

In [ ]:

```
np.linspace(0, 1, 5)
```

In [ ]:

```
np.linspace(0, 1, 20)
```

In [ ]:

```
np.linspace(0, 1, 20, False)
```

---

zeros , ones , empty

In [ ]:

```
np.zeros(5)
```

In [ ]:

```
np.zeros((3, 3))
```

In [ ]:

```
np.zeros((3, 3), dtype=np.int)
```

In [ ]:

```
np.ones(5)
```

In [ ]:

```
np.ones((3, 3))
```

In [ ]:

```
np.empty(5)
```

In [ ]:

```
np.empty((2, 2))
```

identity and eye

In [ ]:

```
np.identity(3)
```

In [ ]:

```
np.eye(3, 3)
```

In [ ]:

```
np.eye(8, 4)
```

In [ ]:

```
np.eye(8, 4, k=1)
```

In [ ]:

```
np.eye(8, 4, k=-3)
```

In [ ]:

```
"Hello World"[6]
```