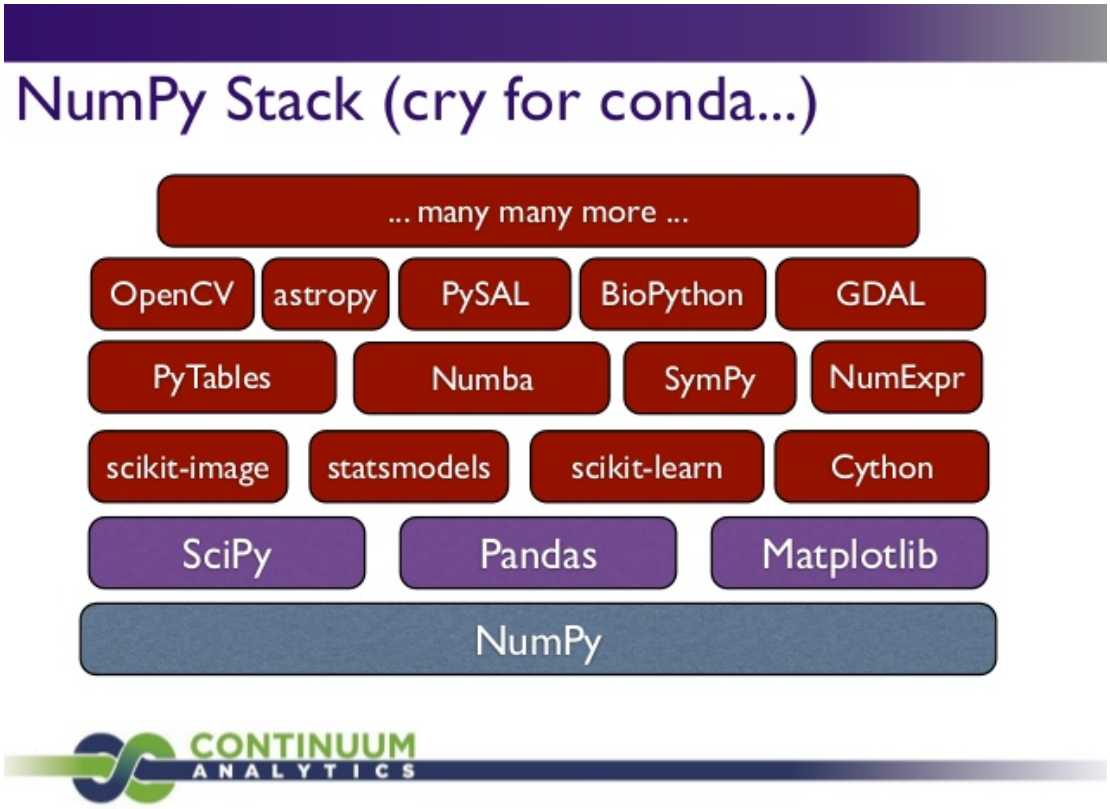
Introduction to numpy

- **ndarray**: the numpy array object
- numerical operations on numpy arrays

The Pydata stack



NumPy and numpy arrays

- The numpy package (module) is used in almost all numerical computation using Python.
- It is a package that provides high-performance vector, matrix and higher-dimensional data structures for Python.
- It is implemented in C and Fortran so when calculations are vectorized (formulated with vectors and matrices), performance is very good.
- Extension package to Python for multi-dimensional arrays

Using Numpy

To use numpy you need to import the module:

```
import numpy as np
```

Creating arrays

1-D array:

In [1]:
```
import numpy as np
a = np.array([0, 1, 2, 3])
a
```

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

In [2]:
```
a.ndim # returns dimension of the array a
```

Out[2]:
```
1
```

In [3]:
```
a.shape # returns the shape of the array a
```

Out[3]:
```
(4,)
```

In [4]:
```
len(a) # returns the length of the array a
```

Out[4]:
```
4
```

Creating 2-D array

In [5]:
```
b = np.array([[0, 1, 2], [3, 4, 5]])    # 2 x 3 array
b
```

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

In [6]:
```
b.ndim
```

Out[6]:
```
2
```

In [7]:
```
b.shape
```

Out[7]:
```
(2, 3)
```

In [8]:
```
len(b)     # returns the size of the first dimension
```

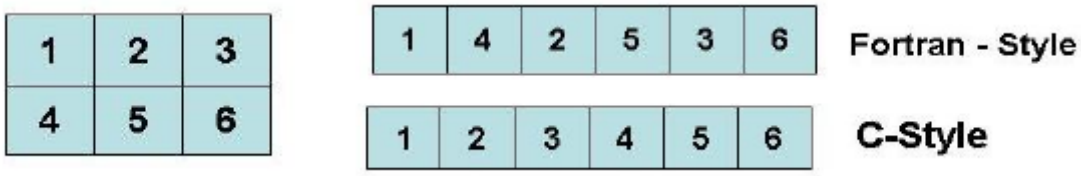
Out[8]:
```
2
```

Basic array creation reference

- The basic array creation function is

```
np.array(object, dtype=None, copy=True, order=None)
```

- `object` is the starting object to convert to np.array, the only mandatory argument
- `dtype` is used to manually specify the type of items
- `copy` if true (default), then the object is copied
- `order` can be 'C' or 'F' according to the preferred allocation row-major or column-major



Functions for creating arrays

In practice, we rarely enter items one by one...

In [9]:
```
# the arange() function return evenly spaced values within a given inter
# used for integers
a = np.arange(10) # 0 .. n-1  (!)
a
```

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

In [10]:
```
b = np.arange(1, 9, 2) # start, end (exclusive), step
b
```

Out[10]:
```
array([1, 3, 5, 7])
```

Functions for creating arrays

In [11]:
```
# the linspace() function return evenly spaced numbers over a specified
# use for floating points
c = np.linspace(0, 1, 6)   # start, end, num-points
c
```

Out[11]:
```
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
```

In [12]:
```
# 5 array items linearly spaced between [0,1)
d = np.linspace(0, 1, 5, endpoint=False)
d
```

Out[12]:
```
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

Functions for creating common arrays

In [13]:
```
b = np.zeros((2, 2))
b
```

Out[13]:
```
array([[ 0.,  0.],
       [ 0.,  0.]])
```

In [14]:
```
d = np.diag(np.array([1, 2, 3, 4]))
d
```

Out[14]:
```
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

Basic data types on array creation

You can explicitly specify which data-type you want:

In [15]:
```
c = np.array([1, 2, 3], dtype=float)
c.dtype
```

Out[15]:
```
dtype('float64')
```

The **default** data type is floating point:

In [16]:
```
a = np.ones((3, 3))
a.dtype
```

Out[16]:
```
dtype('float64')
```

Other basic data types

There are also other types:

In [17]:
```
d = np.array([1+2j, 3+4j, 5+6j]) # dtype Complex
d.dtype
```

Out[17]:
```
dtype('complex128')
```

In [18]:
```
e = np.array([True, False, False, True]) # dtype Bool
e.dtype
```

Out[18]:
```
dtype('bool')
```

In [19]:
```
f = np.array(['Bonjour', 'Hello', 'Hallo',]) # dtype String

f.dtype     # <--- strings containing max. 7 letters
```

Out[19]:
```
dtype('<U7')
```

Much more *data type*: `int8`, `int32`, `int64`, `uint32`, `uint64`, `float16`,...

Indexing and slicing

The items of an array can be accessed and assigned to the same way as other Python sequences (e.g. lists):

In [20]:
```
a = np.arange(10)
a
```

Out[20]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [21]:
```
a[0], a[2], a[-1]
```

Out[21]:
```
(0, 2, 9)
```

In [22]:
```
a=np.arange(10)
a[::-1] #reversing a sequence
```

Out[22]:
```
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

Warning

- Indices begin at 0, like other Python sequences (and C/C++). In contrast, in Fortran or Matlab, indices begin at 1.
- The usual python idiom for reversing a sequence is supported.

Array Indexing

For multidimensional arrays, indexes are tuples of integers:

In [23]:
```
a = np.diag(np.arange(3))
a
```

Out[23]:
```
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 2]])
```

In [24]:
```
a[1, 1]
```

Out[24]:
```
1
```

Array Indexing (2)

In [25]:
```
a[2, 1] = 10 # third line, second column
a
```

Out[25]:
```
array([[ 0,  0,  0],
       [ 0,  1,  0],
       [ 0, 10,  2]])
```

In [26]:
```
a[1] # second line of a
```

Out[26]:
```
array([0, 1, 0])
```

Note that:

- In 2D arrays, the first dimension corresponds to **rows**, the second to **columns**.
- for multidimensional `a`, `a[0]` is interpreted by taking all elements in the unspecified dimensions.

Array Slicing

Arrays, like other Python sequences can also be sliced:

In [27]:
```
a = np.arange(10)
a
```

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

In [28]:
```
a[2:9:3] # [start:end:step]
```

Out[28]:
```
array([2, 5, 8])
```

Note that the first index is not included! :

In [29]:
```
a[:4]
```

Out[29]:
```
array([0, 1, 2, 3])
```

Array Slicing (2)

All three slice components are not required: by default, `start` is 0, `end` is the last and `step` is 1:

In [30]:
```
a[1:3]
```

Out[30]:
```
array([1, 2])
```

In [31]:
```
a[::2]
```

Out[31]:
```
array([0, 2, 4, 6, 8])
```

In [32]:
```
a[3:]
```

Out[32]:
```
array([3, 4, 5, 6, 7, 8, 9])
```

Array Slicing (3)

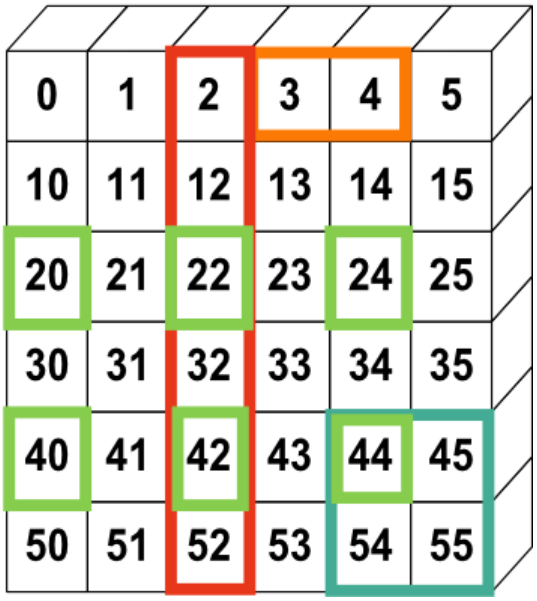
A small illustrated summary of Numpy indexing and slicing...

```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```



Fancy indexing

Numpy arrays can be indexed with slices, but also with boolean or integer arrays (**masks**). This method is called *fancy indexing*.

Using boolean masks

In [33]:
```
a = np.random.randint(0, 20, 15)
a
```

Out[33]:
```
array([ 0,  5,  8,  9, 16, 17,  9, 10, 19,  8,  1, 10, 10, 19, 17])
```

In [34]:
```
(a % 3 == 0)
```

Out[34]:
```
array([ True, False, False,  True, False, False,  True, False, False,
       False, False, False, False, False, False], dtype=bool)
```

In [35]:
```
mask = (a % 3 == 0)
extract_from_a = a[mask] # or,  a[a%3==0]
extract_from_a           # extract a sub-array with the mask
```

Out[35]:
```
array([0, 9, 9])
```

Fancy indexing (2)

Indexing with a mask can be very useful to assign a new value to a sub-array:

In [36]:
```
a[a % 3 == 0] = -1
a
```

Out[36]:
```
array([-1,  5,  8, -1, 16, 17, -1, 10, 19,  8,  1, 10, 10, 19, 17])
```

Indexing with an array of integers

In [37]:
```
a = np.arange(0, 100, 10)
a
```

Out[37]:
```
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Indexing can be done with an array of integers, where the same index is repeated several time:

In [38]:
```
a[[2, 3, 2, 4, 2]]  # note: [2, 3, 2, 4, 2] is a Python list
```

Out[38]:
```
array([20, 30, 20, 40, 20])
```

Fancy indexing (3)

The image below illustrates various fancy indexing applications

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45]])
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
array([2,22,52])
```