NumPy

NumPy

- Is a third-party Python Package (collection of modules)
- Provides fast and efficient tools for numerical/scientific computation

N-dimensional Array

- At the heart of NumPy is a type/class for N-dimensional arrays
  - Arrays are like lists that can only have items of a single type and are of fixed length
  - But more space efficient and with lots of specialized methods
- Type is called `ndarray`

`ndarray` type and `array()`

- The type of values contained in an `ndarray` is decided during creation/instantiation
  - And is contained in the `dtype` attribute of the `ndarray` type
- Straightforward way to create an `ndarray` object is using the numpy `array()` function
  - Accepts an array or sequence type argument

```
>>> import numpy as np

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

>>> print(type(a))
<class 'numpy.ndarray'>

>>> print(a)
[1 2 3]

>>> print(a.dtype)
int64
```

`ndarray` type and `array()`

- Nested sequence as argument generates a multidimensional array
- `ndim:` Number of dimensions in attribute
- `shape`: Size of array in each dimension as a tuple

```
>>> mlist = [[1,2,3],
...          [3,4,5],
...          [6,7,8]]
>>> b = np.array(mlist)

>>> print(b)
[[1 2 3]
 [3 4 5]
 [6 7 8]]

>>> print(a.ndim)
1
>>> print(b.ndim)
2

>>> print(a.shape)
(3,)
>>> print(b.shape)
(3, 3)
```

`dtype` of array

- Above arrays have type of `int64`
  - Signed 64-bit integer
- Automatically determined from most general type in sequence argument
- `dtype` can be explicitly specified at instantiation time
  - Elements are converted to specified type

```
>>> print(a.dtype, b.dtype)
int64 int64

>>> c = np.array([1,2,3,0.2])
>>> print(c.dtype)
float64

>>> d = np.array([1,2,3], dtype=np.float)
>>> print(d)
[ 1.  2.  3.]

>>> print(d.dtype)
float64
```

Other array creation functions

- `arange()`
  Returns array of evenly spaced values of dtype in an interval
- `linspace()`
  Return a specific number of evenly spaced values

```
>>> a = np.arange(10)
>>> print(a)
[0 1 2 3 4 5 6 7 8 9]

>>> print(type(a))
<class 'numpy.ndarray'>

>>> b = np.arange(0,1,0.2)
>>> print(b)
[ 0.   0.2  0.4  0.6  0.8]

>>> b = np.linspace(0,1,8)
>>> print(b)
[ 0.          0.14285714  0.28571429
0.42857143  0.57142857  0.71428571
  0.85714286  1.        ]
>>> b = np.linspace(0,1,15)
>>> print(b)
[ 0.          0.07142857  0.14285714
0.21428571  0.28571429  0.35714286
  0.42857143  0.5         0.57142857
0.64285714  0.71428571  0.78571429
  0.85714286  0.92857143  1.        ]
```

Other array creation functions

- `zeros(), ones(), eye()`
  Arrays initialized with a value
- Take shape argument. Integer or tuple of integers
- `loadtxt()`
  Read row-column oriented data from a file

```
>>> oarr = np.ones((2,2,3), dtype=np.int)

>>> print(oarr)
[[[1 1 1]
  [1 1 1]]

 [[1 1 1]
  [1 1 1]]]

>>> iarr = np.eye(5, dtype=np.int)
>>> print(iarr)
[[1 0 0 0 0]
 [0 1 0 0 0]
 [0 0 1 0 0]
 [0 0 0 1 0]
 [0 0 0 0 1]]
```

`ndarray` methods

- Several methods defined
- Methods may return a copy or a *view*
  E.g. `transpose()` reverses indexes and usually returns a view

```
>>> a = np.random.rand(3,2)
>>> print(a)
[[ 0.67652298  0.75866448]
 [ 0.57583923  0.35294951]
 [ 0.91243552  0.80728729]]
>>> b = a.transpose()
>>> print(b)
[[ 0.67652298  0.57583923  0.91243552]
 [ 0.75866448  0.35294951  0.80728729]]

>>> np.may_share_memory(a,b)
True

>>> b[0][0] = 0.0
>>> print(a)
[[ 0.          0.75866448]
 [ 0.57583923  0.35294951]
 [ 0.91243552  0.80728729]]
```

`ndarray` methods

- `copy()`
  Returns a copy that doesn't share memory
- `fill()`
  Fill the array with a scalar value

ndarray methods

- `reshape()`
  Returns the same array with a new shape (int or tuple). -1 in shape tuple means it is automatically inferred
- `ravel()`
  Return 1-dimensional array

```
>>> mlist = np.array([1,2,3,4,5,6,7,8,9])
>>> print(mlist)
[1 2 3 4 5 6 7 8 9]
>>> print(mlist.shape)
(9,)

>>> mlist = mlist.reshape((3,-1))
>>> print(mlist)
[[1 2 3]
 [4 5 6]
 [7 8 9]]

>>> print(mlist.ravel())
[1 2 3 4 5 6 7 8 9]
```

`ndarray` methods

- Several other methods available

Array Indexing

- Support full Python indexing syntax
- Additionally supports, extended indexing syntax

Array Indexing

- Indexing with tuples

```
>>> mlist = np.array([1,2,3,4,5,6,7,8,9])
>>> mlist = mlist.reshape((3,3))
>>> print(mlist)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> print(mlist[0][2])
3
>>> print(mlist[0,2]) #tuple index
3
```

Array Indexing

- Slice syntax can be combined with tuples
- Unlike lists slices do not return a copy, but a *view*

```
>>> print(mlist[:,::2])
[[1 3]
 [4 6]
 [7 9]]
```

Advanced Indexing

- Use arrays as indexes

```
>>> a = np.array([1,2,3,4,5,6,7])
>>> i = np.array([0,2,2,-1,0,0,-1,3])
>>> print(a[i])
[1 3 3 7 1 1 7 4]

>>> i2 = np.array([[0,0,3,4],
...                [4,4,-1,2]])
>>> print(a[i2])
[[1 1 4 5]
 [5 5 7 3]]
```

Basic Operations

- Mathematical functions are applied element-wise

```
>>> print(a)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> print(b)
[[2 2 2]
 [2 2 2]
 [2 2 2]]
>>> print(a+b)
[[ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>> print(a*b)
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
```

Basic Operations

- `dot()` function
  Dot product or matrix multiplication for 2D

```
>>> print(a)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> print(b)
[[2 2 2]
 [2 2 2]
 [2 2 2]]
>>> print(a+b)
[[ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>> print(a*b)
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
>>> print(np.dot(a,b))
[[12 12 12]
 [30 30 30]
 [48 48 48]]
```

Basic Operations

- Newer Python 3 interpreters support the @ matrix multiplication operator

```
>>> print(a)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> print(b)
[[2 2 2]
 [2 2 2]
 [2 2 2]]
>>> print(np.dot(a,b))
[[12 12 12]
 [30 30 30]
 [48 48 48]]
>>> print(a@b)
[[12 12 12]
 [30 30 30]
 [48 48 48]]
```

SciPy

- Package that contains modules with numerical and scientific tools
- Can be compared to MATLAB toolboxes
- Built on NumPy and operates on NumPy arrays

Example

Linear algebra

- Module must be explicitly imported

```
>>> import scipy.linalg as la

>>> amat = sp.random.random_integers(1,100,
size=(4,4))
>>> bvec = sp.random.random_integers(1,100,
size=4)
>>> print(amat)
[[ 5 99 66 52]
 [55 99 70 85]
 [22 51 73  3]
 [29 15 97 45]]
>>> print(bvec)
[94 81 48 42]

>>> la.det(amat)
-18781401.000000004

>>> x = la.solve(amat,bvec)
>>> print(x)
[-0.44933964  0.60370491  0.36119494
0.24309704]
>>> print(amat@x)
[ 94.  81.  48.  42.]
```

Matplotlib

- Python 2D plotting library
- Large collection of plots
  Line plots, histograms, scatter plots etc.
- Output to screen or “camera-ready” formats
  PDF, TIFF, vector formats etc.

Interface

- MATLAB-like interface called `pylab`
- Object oriented interface

Homework