The numpy module

Working with efficient arrays of values

Arrays of Numbers

- **Arrays** – not built into Python
  - fundamental data structure in computing
- All elements same type
- Elements known to be adjacent to each other in memory
- Regular accesses because of uniformity
  - leads to *faster* access
- Require less memory than lists

The Numpy Module

- Widely used module "numpy" provides an implementation of arrays
  - **np.ndarray** type represents numeric arrays with arbitrary dimensions
  - empty arrays
  - zero-filled arrays
- np.array() method converts Python sequences into arrays
  - converts cube list-of-lists-of-lists to array
  - access with three indexes
    - like the list-of-lists-of-lists approach

Numpy Help

```
import numpy as np
help(np.ndarray)
```

Use numpy with a convenient name…

```
Help on class ndarray in module numpy:

class ndarray(builtins.object)
 |  ndarray(shape, dtype=float, buffer=None, offset=0,
 |          strides=None, order=None)
 |
 |  An array object represents a multidimensional, homogeneous array
 |  of fixed-size items.  An associated data-type object describes the
 |  format of each element in the array (its byte-order, how many bytes it
 |  occupies in memory, whether it is an integer, a floating point number,
 |  or something else, etc.)
 |
 |  Arrays should be constructed using `array`, `zeros` or `empty` (refer
 |  to the See Also section below).  The parameters given here refer to
 |  a low-level method (`ndarray(...)`) for instantiating an array.
 |
 |  For more information, refer to the `numpy` module and examine the
 |  methods and attributes of an array.
 |
 |  Parameters
```

np.ndarray Arrays

```
triple = np.zeros( [3] )
square = np.zeros( [2, 2] )
line = np.empty( [8] )

print(triple)
print()
print(square)
print()
print(line)
```

```
[ 0.  0.  0.]

[[ 0.  0.]
 [ 0.  0.]]

[  6.90865611e-310   4.68012469e-310   0.00000000e+000   0.00000000e+000
   0.00000000e+000   0.00000000e+000   0.00000000e+000               nan]
```

- 1-dimensional, 3-element array of floating-point zeros
- 2x2-element square array of floating-point zeros
- 8-element array of floating-points, uninitialized

Generate Numpy Arrays

- numpy alternatives to "range()" function:
  - `vec1 = np.arange(start, stop, step)`
    - array is half-open interval, excludes "`stop`" value
    - Similar to python "range()" – not good for real #'s
  - `vec2 = np.linspace(start, end, nsamples)`
    - array is closed interval, includes both "`start`" and "`end`"
    - Best for non-integer intervals
- Change an array's shape:
  - `a1 = vec1.reshape( shape-tuple )`
  - `shape-tuple` argument defines new array's shape
    - number of elements gives array's dimensions
    - element values give dimension extents
      - example: (6, 6, 6) – three dimensions, 6 elements in each

Convert Python Lists to Numpy Arrays

- 1-dimensional list to array:
  - ```
    mylist1 = [ x**2 for x in range(100) ]

    myarray1 = np.array(mylist1)
    print(myarray1.shape, myarray1.ndim)
    ```

- 2-dimensional list-of-lists to array:
  - ```
    mylist2 = [  \
          [ x**2/y for x in range(50) ]    \
               for y in range(1,10) ]

    myarray2 = np.array( mylist2 )
    print(myarray2.shape, myarray2.ndim)
    ```

Array-Making Functions

- np.zeros(shape-tuple)
  np.ones(shape-tuple)
  - create arrays filled with 0s or with 1s
- np.empty(shape-tuple)
  - create an array with no particular values
- shape-tuple argument
  - defines shape of the array
  - number of elements gives array's dimensions
  - element values give dimension extents
    - ex. (4, 4, 4) – three dimensions, 4 elements in each

More Functions to Make Arrays Directly

- np.full(shape-tuple , fill-value)
  - create an array filled with any desired value

Create an array shaped like a given pre-existing array:

- np.zeros_like(array)
  np.ones_like(array)
  np.empty_like(array)
  np.full_like(array, fill-value)

numpy Versus math

- "math" module provides advanced, floating-point functions, for example:
  - math.log, math.exp
  - math.pi, math.e
  - math.sin, math.cos, math.tan
- "numpy" module provides comparable functions, extended to apply to ndarrays
- Also, related submodules/functions:
  - np.random.random(), np.random.random_integer()
  - (More functions in np.random)

Random Arrays; Array-Oriented Actions

- Create arrays of random values
- Math operators apply to arrays, and *scalars*

```
sq1 = np.random.randint(0,11, (2,2))
sq2 = 10 * np.random.random((2,2,))

print('sq1:\n', sq1, '\nsq2:\n', sq2)

print('\nproduct:\n', np.round(sq1 * sq2, 2) )
```

scalar w/ array

```
sq1:
 [[2 6]
 [2 0]]
sq2:
 [[8.39034424 8.10763239]
 [2.62528707 9.84959535]]

product:
 [[16.78 48.65]
 [ 5.25  0.  ]]
```

array - array product; rounding the results

Multiplying Arrays

- Two arrays can be multiplied element-by-element
  - Also: add, subtract, divide, other operations
- Linear/matrix algebra
  - Numpy provides .dot(), .cross(), .diagonal() functions
  - Array shapes must be compatible
  - Usage is beyond the scope of this introduction

Reduction Operations

- "Collapse arrays to fewer dimensions or to scalars

```
cube = np.array([ [1,2,3], [0.5,-7.3,4.44] ])
print(cube.shape, '\n', cube, '\n#--------')
print('Column-wise sums:', np.sum(cube, axis=0), '\n#--------')
print('   Row-wise sums:', np.sum(cube, axis=1), '\n#--------')
print('    Overall sum: ', np.sum(cube), '\n#--------')
```

```
(2, 3)
 [[ 1.    2.    3.  ]
 [ 0.5  -7.3   4.44]]
#--------
Column-wise sums: [ 1.5  -5.3   7.44]
#--------
   Row-wise sums: [ 6.   -2.36]
#--------
    Overall sum:  3.6400000000000006
#--------
```

sum() function reduces array's dimensionality

Example: powers of 2

```
# Naive, pythonic approach:
exponents = np.empty( (8,) )
for i in range(8):
    exponents[i] = i

exponents    # floating-point!
```

```
array([0., 1., 2., 3., 4., 5., 6., 7.])
```

```
# Numpy approach:
exponents = np.arange(8)

exponents    # integer
```

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
# Numpy approach 2:
exponents = np.linspace(0, 7, 8)

exponents    # floating-point again
```

```
array([0., 1., 2., 3., 4., 5., 6., 7.])
```

```
powers = 2 ** exponents
powers
```

```
array([  1.,   2.,   4.,   8.,  16.,  32.,  64., 128.])
```

Example: trig function

```
x = np.linspace(0, 6*np.pi, 100)
y = np.sin(x) * np.cos(3*x) * np.exp(-.1 * x)
```

```
import matplotlib.pyplot as plt
plt.plot(x,y)
plt.show()
```