Numerical and Scientific Packages

Dr. Shailesh Sivan

Numeric and scientific applications

- As you might expect, there are a number of third-party packages available for numerical and scientific computing that extend Python's basic math module.
- These include:
- NumPy/SciPy – numerical and scientific function libraries.
- Numba – Python compiler that support JIT compilation.
- ALGLIB – numerical analysis library.
- Pandas – high-performance data structures and data analysis tools.
- PyGSL – Python interface for GNU Scientific Library.
- ScientificPython – collection of scientific computing modules.

Scipy and friends

- By far, the most commonly used packages are those in the SciPy stack. We will focus on these in this class. These packages include:
- NumPy
- SciPy
- Matplotlib – plotting library.
- IPython – interactive computing.
- Pandas – data analysis library.
- SymPy – symbolic computation library.

Numpy

- Let's start with NumPy. Among other things, NumPy contains:
- A powerful N-dimensional array object.
- Sophisticated functions.
- Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data.

Numpy

- The key to NumPy is the ndarray object, an *n*-dimensional array of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:
- NumPy arrays have a fixed size. Modifying the size means creating a new array.
- NumPy arrays must be of the same data type, but this can include Python objects.
- More efficient mathematical operations than built-in sequence types.

NumPy arrays

- There are a couple of mechanisms for creating arrays in NumPy:
- Conversion from other Python structures (e.g., lists, tuples).
- Built-in NumPy array creation (e.g., arange, ones, zeros, etc.).
- Reading arrays from disk, either from standard or custom formats (e.g. reading in from a CSV file).
- and others …

NumPy arrays

- In general, any numerical data that is stored in an array-like container can be converted to an ndarray through use of the array() function. The most obvious examples are sequence types like lists and tuples.

```
>>> x = np.array([2,3,1,0])
>>> x = np.array([2, 3, 1, 0])
>>> x = np.array([[1,2.0],[0,0],(1+1j,3.)])
>>> x = np.array([[ 1.+0.j, 2.+0.j], [ 0.+0.j, 0.+0.j], [ 1.+1.j, 3.+0.j]])
```


NumPy arrays

- There are a couple of built-in NumPy functions which will create arrays from scratch.
- zeros(shape) -- creates an array filled with 0 values with the specified shape. The default dtype is float64

```
>>> np.zeros((2, 3))
array([[ 0., 0., 0.], [ 0., 0., 0.]])
```

- ones(shape) -- creates an array filled with 1 values.
- arange() -- creates arrays with regularly incrementing values.

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(2, 10, dtype=np.float)
array([ 2., 3., 4., 5., 6., 7., 8., 9.])
>>> np.arange(2, 3, 0.1)
array([ 2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])
```

NumPy arrays

- linspace() -- creates arrays with a specified number of elements, and spaced equally between the specified beginning and end values.

```
>>> np.linspace(1., 4., 6)
array([ 1. , 1.6, 2.2, 2.8, 3.4, 4. ])
```

- random.random(shape) – creates arrays with random floats over the interval [0,1).

```
>>> np.random.random((2,3))
array([[ 0.75688597, 0.41759916, 0.35007419],
       [ 0.77164187, 0.05869089, 0.98792864]])
```


NumPy arrays

- Printing an array can be done with the print statement.

```
>>> import numpy as np
>>> a = np.arange(3)
>>> print(a)
[0 1 2]
>>> a
array([0, 1, 2])
>>> b = np.arange(9).reshape(3,3)
>>> print(b)
[[0 1 2]
 [3 4 5]
 [6 7 8]]
>>> c = np.arange(8).reshape(2,2,2)
>>> print(c)
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

Indexing

- Single-dimension indexing is accomplished as usual.

```
>>> x = np.arange(10)
>>> x[2]
2
>>> x[-2]
8
```

0 1 2 3 4 5 6 7 8 9

- Multi-dimensional arrays support multi-dimensional indexing.

```
>>> x.shape = (2,5) # now x is 2-dimensional
>>> x[1,3]
8
>>> x[1,-1]
9
```

0 1 2 3 4
5 6 7 8 9

Indexing

- Using fewer dimensions to index will result in a subarray.

```
>>> x[0]
array([0, 1, 2, 3, 4])
```

- This means that `x[i, j] == x[i][j]` but the second method is less efficient.

Indexing

- Slicing is possible just as it is for typical Python sequences.

```
>>> x = np.arange(10)
>>> x[2:5]
array([2, 3, 4])
>>> x[:-7]
array([0, 1, 2])
>>> x[1:7:2]
array([1, 3, 5])
>>> y = np.arange(35).reshape(5,7)
>>> y[1:5:2,::3]
array([[ 7, 10, 13], [21, 24, 27]])
```


Array operations

```
>>> a = np.arange(5)
>>> b = np.arange(5)
>>> a+b
array([0, 2, 4, 6, 8])
>>> a-b
array([0, 0, 0, 0, 0])
>>> a**2
array([ 0,  1,  4,  9, 16])
>>> a>3
array([False, False, False, False,  True], dtype=bool)
>>> 10*np.sin(a)
array([ 0., 8.41470985, 9.09297427, 1.41120008, -7.56802495])
>>> a*b
array([ 0,  1,  4,  9, 16])
```

- Basic operations apply element-wise. The result is a new array with the resultant elements.

  Operations like *= and += will modify the existing array.

Array operations

- Since multiplication is done element-wise, you need to specifically perform a dot product to perform matrix multiplication.

```
>>> a = np.zeros(4).reshape(2,2)
>>> a
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> a[0,0] = 1
>>> a[1,1] = 1
>>> b = np.arange(4).reshape(2,2)
>>> b
array([[0, 1],
       [2, 3]])
>>> a*b
array([[ 0.,  0.],
       [ 0.,  3.]])
>>> np.dot(a,b)
array([[ 0.,  1.],
       [ 2.,  3.]])
```


Array operations

- There are also some built-in methods of ndarray objects.

  Universal functions which may also be applied include exp, sqrt, add, sin, cos, etc…

```
>>> a = np.random.random((2,3))
>>> a
array([[ 0.68166391,  0.98943098,  0.69361582],
       [ 0.78888081,  0.62197125,  0.40517936]])
>>> a.sum()
4.1807421388722164
>>> a.min()
0.4051793610379143
>>> a.max(axis=0)
array([ 0.78888081,  0.98943098,  0.69361582])
>>> a.min(axis=1)
array([ 0.68166391,  0.40517936])
```

Array operations

- An array shape can be manipulated by a number of methods.

  resize(size) will modify an array in place.

  reshape(size) will return a copy of the array with a new shape.

```
>>> a = np.floor(10*np.random.random((3,4)))
>>> print(a)
[[ 9. 8. 7. 9.]
 [ 7. 5. 9. 7.]
 [ 8. 2. 7. 5.]]
>>> a.shape
(3, 4)
>>> a.ravel()
array([ 9., 8., 7., 9., 7., 5., 9., 7., 8., 2., 7., 5.])
>>> a.shape = (6,2)
>>> print(a)
[[ 9. 8.]
 [ 7. 9.]
 [ 7. 5.]
 [ 9. 7.]
 [ 8. 2.]
 [ 7. 5.]]
>>> a.transpose()
array([[ 9., 7., 7., 9., 8., 7.],
       [ 8., 9., 5., 7., 2., 5.]])
```