AMath 483/583 — Lecture 26 — May 25, 2011

Today:

- NumPy arrays
- Python functions
- Iterators, booleans

Next:

- Python exception handling
- Python plus Fortran

Read: Class notes and references

Lists

The elements of a list can be any objects
(need not be same type):

```
>>> L = [3, 4.5, 'abc', [1,2]]
```

Indexing starts at 0:

```
>>> L[0]
3

>>> L[2]
'abc'

>>> L[3]
[1, 2]

>>> L[3][0]  # element 0 of L[3]
1
```

Lists aren't good as numerical arrays

Lists in Python are quite general, can have arbitrary objects as elements.

Addition and scalar multiplication are defined for lists, but not what we want for numerical computation, e.g.

Multiplication repeats:

```
>>> x = [2., 3.]
>>> 2*x
[2.0, 3.0, 2.0, 3.0]
```

Addition concatenates:

```
>>> y = [5., 6.]
>>> x+y
[2.0, 3.0, 5.0, 6.0]
```

NumPy module

Instead, use NumPy arrays:

```
>>> import numpy as np

>>> x = np.array([2., 3.])
>>> 2*x
array([ 4.,  6.])

>>> y = np.array([5., 6.])
>>> x+y
array([ 7.,  9.])
```

Other operations also apply component-wise:

```
>>> np.sqrt(x)
array([ 1.41421356,  1.73205081])
>>> np.cos(x)
array([-0.41614684, -0.9899925 ])
```

NumPy arrays

Unlike lists, all elements of an `np.array` have the same type

```
>>> np.array([1, 2, 3])     # all integers
array([1, 2, 3])

>>> np.array([1, 2, 3.])    # one float
array([ 1.,  2.,  3.])      # they're all floats!
```

Can explicitly state desired data type:

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

>>> (x + 1.j) * 2.j
array([-2.+2.j, -2.+4.j, -2.+6.j])
```

NumPy arrays for vectors and matrices

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

>>> A.shape
(3, 2)

>>> A.T
array([[ 1.,  3.,  5.],
       [ 2.,  4.,  6.]])

>>> x = np.array([1., 1.])
>>> x.T
array([ 1.,  1.])
```

NumPy arrays for vectors and matrices

Can index into multi-dimensional arrays:

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


>>> A[1,0]
3.0
```

Better than as list of lists...

```
>>> A[1][0]
3.0
```

NumPy arrays for vectors and matrices

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

>>> x
array([ 1.,  1.])

>>> np.dot(A,x)      # matrix-vector product
array([  3.,   7.,  11.])

>>> np.dot(A.T, A)    # matrix-matrix product
array([[ 35.,  44.],
       [ 44.,  56.]])
```

NumPy matrices for vectors and matrices

For Linear algebra, may instead want to use numpy.matrix:

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

Or, Matlab style (as a string that is converted):

```
>>> A = np.matrix("1.,2; 3,4; 5,6")
>>> A
matrix([[ 1.,  2.],
        [ 3.,  4.],
        [ 5.,  6.]])
```

NumPy matrices for vectors and matrices

Note: vectors are handled as matrices with 1 row or column:

```
>>> x = np.matrix("4.;5.")
>>> x
matrix([[ 4.],
        [ 5.]])
>>> x.T
matrix([[ 4.,  5.]])
>>> A*x
matrix([[ 14.],
        [ 32.],
        [ 50.]])
```

But note that indexing into x requires two indices:

```
>>> print x[0,0], x[1,0]
4.0 5.0
```

Which to use, array or matrix?

For linear algebra matrix may be easier (and more like Matlab), but vectors need two subscripts!

For most other uses, arrays more natural, e.g.

```
>>> x = np.linspace(0., 3., 100)  # 100 points
>>> y = x**5 - 2.*sqrt(x)*cos(x)  # 100 values
>>> plot(x,y)
```

np.linspace returns an array, which is what is needed here.

We will always use arrays. If you want to specify Matlab-style:

```
>>> B = np.matrix("1,2; 3,4").A
>>> B
array([[1, 2],
       [3, 4]])
```

Rank of an array

The rank of an array is the number of subscripts it takes:

```
>>> A = np.ones((4,4))

>>> A
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])

>>> np.rank(A)
2
```

Warning: This is not the rank of the matrix in the linear algebra sense (dimension of the column space)!

Rank of an array

Scalars have rank 0:

```
>>> z = np.array(7.)
>>> z
array(7.0)
```

NumPy arrays of any dimension are supported, e.g. rank 3:

```
>>> T = np.ones((2,2,2))
>>> T
array([[[ 1.,  1.],
        [ 1.,  1.]],

       [[ 1.,  1.],
        [ 1.,  1.]]])
>>> T[0,0,0]
1.0
```

Linear algebra with NumPy

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

>>> b = np.dot(A, np.array([8., 9.]))
>>> b
array([ 26.,  60.])
```

Now solve $Ax = b$:

```
>>> from numpy.linalg import solve
>>> solve(A,b)
array([ 8.,  9.])
```

Eigenvalues

```
>>> from numpy.linalg import eig

>>> eig(A)  # returns a tuple (evals,evecs)

(array([-0.37228132,  5.37228132]),
       array([[-0.82456484, -0.41597356],
              [ 0.56576746, -0.90937671]]))

>>> evals, evecs = eig(A)  # unpacks tuple

>>> evals
array([-0.37228132,  5.37228132])

>>> evecs
array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]])
```

Function inputs and outputs

A sample function with 1 required input, 2 optional inputs, and 2 outputs:

```
>>> def f(x,y=2,z=None):
...     val = x+y
...     if z:
...         val = val + z
...     return val,y   # or return (val,y)
...
>>> f(1)
(3, 2)

>>> f(1,6,10)
(17, 6)

>>> f(1,z=5)
(8, 2)
```

Function inputs and outputs

f returns two values, so we could also do:

```
>>> v1, v2 = f(1,z=5)
>>> v1
8
>>> v2
2
```

x has no default value, so it must always be specified:

```
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes at least
           1 argument (0 given)
```

A function with no output

This function does not return any value:

```
>>> def f2(x):
...     y = x**2
...     print "The square is ", y
...

>>> f2(3)
The square is  9

>>> y = f2(100)
The square is  10000

>>> print y
None
```

Note: None is a special pre-defined object in Python

Quadrature (numerical integration)

Estimate $\int_0^2 x^2\,dx = 8/3$:

```
>>> from scipy.integrate import quad

>>> def f(x):
...     return x**2
...
>>> quad(f, 0., 2.)
(2.6666666666666667, 2.9605947323337514e-14)
```

returns (value, error estimate).

Other keyword arguments to set error tolerance, for example.

Lambda functions

In the last example, f is so simple we might want to just include its definition directly in the call to quad.

We can do this with a lambda function:

```
>>> f = lambda x: x**2
>>> f(4)
16
```

This defines the same f as before. But instead we could do:

```
>>> quad(lambda x: x**2, 0., 2.)
(2.666666666666667, 2.960594732333751e-14)
```

Random numbers

```
In [36]: from numpy.random import uniform
In [37]: uniform?

Docstring:
    uniform(low=0.0, high=1.0, size=1)
    Draw samples from a uniform distribution.
    etc.

In [38]: uniform()
Out[38]: 0.052044690516633407

In [39]: uniform(size=(2,3)) # NOTE: keyword arg.
Out[39]:
array([[ 0.95581274,  0.07874926,  0.30454462],
       [ 0.53318419,  0.27670149,  0.16840566]])
```

Python loops

Example: iterating over items of a list

```
for j in [1,2,3]:
    print "j is now ", j
    print "this is also in loop"
print "the loop has ended"
```

produces:

```
j is now  1
this is also in loop
j is now  2
this is also in loop
j is now  3
this is also in loop
the loop has ended
```

Remember: Indentation determines what's in the loop.

Python loops

More generally:

```
for j in some_iterable_object:
    # contents of loop
```

Certain types or classes of objects are iterable.
Requires a pre-defined way to loop over the contents.

Lists and tuples are iterable, as in last example.

Strings are iterable:

```
for j in 'abc':
    print "j is now ", j
```

produces:

```
j is now  a
j is now  b
j is now  c
```

Python loops

To loop over the indices rather than over the values, use:

```
L = [12,5,7]
for j in range(len(L)):
    print "j = %s, L[%s] is %s" % (j,j,L[j])
```

produces

```
j = 0, L[0] is 12
j = 1, L[1] is 5
j = 2, L[2] is 7
```

Note that:

```
>>> len(L)
3
>>> range(3)
[0, 1, 2]
```

So range(len(L)) produces a list with the proper indices.

enumerate

Another way to do this is with enumerate:

```
L = [12,5,7]
for j,value in enumerate(L):
    print "L[%s] is %s" % (j,value)
```

also produces

```
L[0] is 12
L[1] is 5
L[2] is 7
```

Python if-then-else

```
x = 2
if x < 1:
    print "x is less than 1"
elif x > 3:
    print "x is larger than 3"
else:
    print "x is between 1 and 3"
```

produces:

```
x is between 1 and 3
```

Note indentation!

There can be 1 or more elif (else if) statements, or none.

The else is also optional.

Booleans

An object of type bool has value True or False.

```
>>> x = 2
>>> large = x > 100

>>> large
False
>>> type(large)
<type 'bool'>
```

Any boolean can go in the test of the if or elif clauses.

Note: To test equality, use == and for inequality, !=

```
>>> x==2
True
>>> x!=2
False
```

Booleans

Can use:

and or &

or or |

```
>>> x = 0.5
>>> ((x>3) and (x<4)) or ((x>0) and (x<1))
True
```

is the same as

```
>>> ((x>3) & (x<4)) | ((x>0) & (x<1))
True
```

Booleans

Other objects can also go in the test of the if or elif clauses.

In general, object acts as True unless it is:

- a boolean with value False,
- a integer with value 0, float with value 0.,
- an empty list [ ], empty string "", etc.
- the special object None.

```
y = None
if y:
    print "y is ", y
else:
    y = 0.
    print "initialized y"
```

produces:

```
initialized y
```

Note: If y is not defined at all this will give an error.
(i.e., raise an exception)