

Physics 514 – Basic Python Intro, Part III

Emanuel Gull

September 6, 2016

1 Introduction

This is part III of the python introduction: Introduction to Scipy and Numpy, essential numerical methods, simple plotting.

2 Installing Numpy and Scipy

Run python, check if you have the packages:

```
egull$ python
Python 2.7.3 (default, Apr 19 2012, 00:55:09)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> import scipy
>>>
```

If you see no error messages, scipy has been imported successfully. Otherwise you'll see something like this:

```
egull$ /usr/bin/python
Python 2.7.1 (r271:86832, Aug 5 2011, 03:30:24)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import scipy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named scipy
>>>
```

If so: go to <http://www.scipy.org/Download> and download numpy and scipy.

3 Linear Algebra

Most examples are taken from the tentative numpy tutorial:
http://scipy.org/Tentative_NumPy_Tutorial.

3.1 Vector and matrix creation

```
>>> import numpy
>>> a=numpy.arange(15).reshape(3,5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.size
15
>>> type(a)
<type 'numpy.ndarray'>
>>> b=numpy.array([2,3,4])
>>> b
array([2, 3, 4])
```

Numpy automatically creates arrays with a convenient type:

```
a=numpy.array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> a=numpy.array([2,3,4.])
>>> a.dtype
dtype('float64')
>>> a=numpy.array([2,3,4.j])
>>> a.dtype
dtype('complex128')
>>> a=numpy.array([2,3,"4"])
>>> a.dtype
dtype('|S1')
>>> a[0]
'2'
```

You can also explicitly force the type:

```
>>> a=numpy.array([2,3,"4"], dtype='float64')
>>> a
array([ 2.,  3.,  4.])
>>> a.dtype
dtype('float64')
```

3.2 Numpy functions

The function `linspace` is often useful to create linearly spaced meshes:

```
>>> x=numpy.linspace( 0, 2*numpy.pi, 100 )
>>> y=numpy.cos(x)
>>> y
```

This creates a range of linearly spaced numbers between zero and 100. The second call computes the numpy cos function for the entire array. Careful: there are math functions in `numpy` and math functions in `math`. If you use the wrong one:

```
>>> import math
>>> y=math.cos(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars

'math' won't know how to operate on numpy arrays.
Other useful functions:  dir(numpy)!
```

3.3 Vector and matrix manipulation

Create a vector and a matrix, multiply them

```
>>> A=numpy.random.random(20).reshape(5,4)
>>> A
array([[ 0.06035581,  0.22310825,  0.66284172,  0.8157386 ],
       [ 0.7596335 ,  0.73138406,  0.65045423,  0.99820784],
       [ 0.58002087,  0.0490578 ,  0.64295104,  0.32102289],
       [ 0.54685319,  0.89919798,  0.27781809,  0.07497537],
       [ 0.98935254,  0.93110438,  0.68877384,  0.70859095]])
>>> A.transpose()
array([[ 0.06035581,  0.7596335 ,  0.58002087,  0.54685319,  0.98935254],
       [ 0.22310825,  0.73138406,  0.0490578 ,  0.89919798,  0.93110438],
       [ 0.66284172,  0.65045423,  0.64295104,  0.27781809,  0.68877384],
       [ 0.8157386 ,  0.99820784,  0.32102289,  0.07497537,  0.70859095]])
>>>
>>> B=numpy.random.random(4)
>>> numpy.dot(A,B)
array([ 0.21717056,  1.04687547,  0.49993608,  0.92030979,  1.31887798])
>>> C=numpy.random.random(5)
>>> numpy.dot(C,A)
array([ 1.87220465,  1.82494789,  1.76809095,  1.67655998])
```

Element and row/column addressing

```
>>> A[1,1]
0.7313840588852598
>>> A[1,: ]
array([ 0.7596335 ,  0.73138406,  0.65045423,  0.99820784])
```

```
>>> A[:,2:3]
array([[ 0.66284172],
       [ 0.65045423],
       [ 0.64295104],
       [ 0.27781809],
       [ 0.68877384]])
>>>
```

Trace calculation and determinant

```
>>> A.trace()
1.5096662828176592
>>> D=numpy.random.random([5,5])
>>> D
array([[ 0.77481702,  0.95690445,  0.38370375,  0.66766355,  0.4503318 ],
       [ 0.93140967,  0.65364293,  0.85593413,  0.86780891,  0.98145262],
       [ 0.19453299,  0.25061024,  0.94314815,  0.01604057,  0.89886711],
       [ 0.07582915,  0.47058326,  0.23647779,  0.55750474,  0.79162348],
       [ 0.87564761,  0.89671009,  0.3409372 ,  0.7866741 ,  0.75093861]])
>>> numpy.linalg.det(D)
-0.069272897280559145
```

Exercise: Write a routine to create two random square matrices of size 100. Multiply them using nested for loops and time your result (repeat it to get statistics, so that the simulation takes around 5 seconds). Do the same with `numpy.dot`. Compare the timings, prepare a plot of the time per matrix multiplication as a function of matrix size, e-mail it to egull@umich.edu.

There are many ways of storing data on disk. One of the most convenient ones:

```
>>> A=numpy.random.random([20,40])
>>> numpy.savetxt("A.dat", A)
```

...and to reload:

```
>>> B=numpy.loadtxt("A.dat")
>>> B.shape
(20, 40)
```

3.4 Equation solving, matrix decomposition

We have already used the determinant function from the `numpy.linalg` package. Here are some more examples:

```
>>> D
array([[ 0.77481702,  0.95690445,  0.38370375,  0.66766355,  0.4503318 ],
       [ 0.93140967,  0.65364293,  0.85593413,  0.86780891,  0.98145262],
       [ 0.19453299,  0.25061024,  0.94314815,  0.01604057,  0.89886711],
```

```

        [ 0.07582915, 0.47058326, 0.23647779, 0.55750474, 0.79162348],
        [ 0.87564761, 0.89671009, 0.3409372 , 0.7866741 , 0.75093861]])
>>> A
array([[ 0.06035581, 0.22310825, 0.66284172, 0.8157386 ],
       [ 0.7596335 , 0.73138406, 0.65045423, 0.99820784],
       [ 0.58002087, 0.0490578 , 0.64295104, 0.32102289],
       [ 0.54685319, 0.89919798, 0.27781809, 0.07497537],
       [ 0.98935254, 0.93110438, 0.68877384, 0.70859095]])
>>> F=numpy.linalg.solve(D,A)
>>> F
array([[ 1.79100105, 0.57586965, 0.36475599, 0.47130924],
       [-0.84111143, -0.60561888, 0.61809185, 0.11494485],
       [-1.97745215, -1.54582183, 0.11879924, 0.7471441 ],
       [-1.39434182, 0.26477572, -0.62102513, 0.46459401],
       [ 2.59192886, 1.71604711, 0.3504524 , -0.56914819]])
>>> numpy.dot(D,F)
array([[ 0.06035581, 0.22310825, 0.66284172, 0.8157386 ],
       [ 0.7596335 , 0.73138406, 0.65045423, 0.99820784],
       [ 0.58002087, 0.0490578 , 0.64295104, 0.32102289],
       [ 0.54685319, 0.89919798, 0.27781809, 0.07497537],
       [ 0.98935254, 0.93110438, 0.68877384, 0.70859095]])
>>> Aprime=numpy.dot(D,F)
>>> numpy.max(numpy.abs((A-Aprime).reshape(-1)))
2.2204460492503131e-16
>>>

```

Similarly useful are decompositions: *LU*, *QR*, *SVD* (singular value decomposition), etc: This is an example of a *QR* decomposition:

```

>>> D
array([[ 0.77481702, 0.95690445, 0.38370375, 0.66766355, 0.4503318 ],
       [ 0.93140967, 0.65364293, 0.85593413, 0.86780891, 0.98145262],
       [ 0.19453299, 0.25061024, 0.94314815, 0.01604057, 0.89886711],
       [ 0.07582915, 0.47058326, 0.23647779, 0.55750474, 0.79162348],
       [ 0.87564761, 0.89671009, 0.3409372 , 0.7866741 , 0.75093861]])
>>> Q,R=numpy.linalg.qr(D)
>>> Q
array([[ 0.51333583, 0.38989112, -0.19667744, -0.43425811, -0.59766974],
       [ 0.61708242, -0.49026219, 0.25245345, 0.50689728, -0.24119426],
       [ 0.12888301, 0.11789435, 0.88293112, -0.39501948, 0.18407158],
       [ 0.05023872, 0.76594938, 0.15917624, 0.61954638, 0.04028443],
       [ 0.58013865, 0.08396648, -0.30443523, -0.12081779, 0.74102027]])
>>> numpy.linalg.det(Q)
-0.9999999999999998
>>> R
array([[ 1.50937645, 1.47072198, 1.05637775, 1.36470097, 1.42807597],
       [ 0.          , 0.5179142 , 0.05091969, 0.32982794, 0.46977979],

```

```

    [ 0.          , 0.          , 0.90720086, -0.04882009, 0.85023421],
    [ 0.          , 0.          , 0.          , 0.39397114, 0.34658612],
    [ 0.          , 0.          , 0.          , 0.          , 0.24793629]])
>>> numpy.dot(Q,R)
array([[ 0.77481702,  0.95690445,  0.38370375,  0.66766355,  0.4503318 ],
       [ 0.93140967,  0.65364293,  0.85593413,  0.86780891,  0.98145262],
       [ 0.19453299,  0.25061024,  0.94314815,  0.01604057,  0.89886711],
       [ 0.07582915,  0.47058326,  0.23647779,  0.55750474,  0.79162348],
       [ 0.87564761,  0.89671009,  0.3409372 ,  0.7866741 ,  0.75093861]])
>>>

```

An example of an SVD: S

```

>>> U,Sigma,Vstar=numpy.linalg.svd(A)
>>> S=numpy.zeros([5,4])
>>> S[:4,:4]=numpy.diag(Sigma)
>>> numpy.dot(numpy.dot(U,S),Vstar)
array([[ 0.06035581,  0.22310825,  0.66284172,  0.8157386 ],
       [ 0.7596335 ,  0.73138406,  0.65045423,  0.99820784],
       [ 0.58002087,  0.0490578 ,  0.64295104,  0.32102289],
       [ 0.54685319,  0.89919798,  0.27781809,  0.07497537],
       [ 0.98935254,  0.93110438,  0.68877384,  0.70859095]])
>>> S
array([[ 2.72849906,  0.          ,  0.          ,  0.          ],
       [ 0.          ,  0.89474705,  0.          ,  0.          ],
       [ 0.          ,  0.          ,  0.49562301,  0.          ],
       [ 0.          ,  0.          ,  0.          ,  0.25831974],
       [ 0.          ,  0.          ,  0.          ,  0.          ]])
>>> numpy.linalg.det(U), numpy.linalg.det(Vstar)
(0.9999999999999999, 0.9999999999999994)
>>> U
array([[ -0.31882981,  0.65617049,  0.38697503, -0.5468781 ,  0.13770443],
       [ -0.57683841,  0.1592713 ,  0.23141527,  0.558861 , -0.52536793],
       [ -0.2896256 ,  0.22967905, -0.86789488, -0.22994484, -0.23926637],
       [ -0.33131293, -0.65627096,  0.17925868, -0.55459148, -0.34617161],
       [ -0.60988012, -0.24622936, -0.10640538,  0.16778673,  0.72659676]])
>>> Vstar
array([[ -0.51676173, -0.50321085, -0.47090724, -0.5079198 ],
       [ -0.34499288, -0.60936738,  0.3736117 ,  0.60833018],
       [ -0.62849062,  0.55511602, -0.3520289 ,  0.41583819],
       [ 0.4679068 , -0.25941557, -0.71745127,  0.44612831]])

```

3.5 Eigensystems

```

>>> w,v=numpy.linalg.eig(D)
>>> numpy.dot(D, v[:,0])-w[0]*v[:,0]
array([ -3.33066907e-16 +3.88578059e-16j,

```

```
1.66533454e-16 -2.35922393e-16j,
1.66533454e-16 -6.10622664e-16j, 2.22044605e-16 +3.33066907e-16j])
```

It is much cheaper to only evaluate the eigenvalues (not the eigenvectors). `eigenvals` does this:

```
>>> numpy.linalg.eigvals(D)
array([ 0.48319792+0.87551115j, 0.48319792-0.87551115j,
       -0.99921277+0.03967177j, -0.99921277-0.03967177j])
>>> w
array([ 0.48319792+0.87551115j, 0.48319792-0.87551115j,
       -0.99921277+0.03967177j, -0.99921277-0.03967177j])
```

Exercise: Solve a linear equation using `solve`, and by back-substituting into an LU decomposed matrix

4 Quadrature

Exercise: Program a Simpson integration routine or use the one of last week, integrate `sin` from 0 to π .

Python provides a set of general purpose quadrature algorithms, both for function objects and for sampled functions. `quad` is the all-purpose integrator for a function:

```
>>> scipy.integrate.quad(numpy.sin,0,numpy.pi)
(2.0, 2.220446049250313e-14)
```

The same works for 2d integration (see the python book for the lambda notation. In this context 'lambda' creates an inline function that always returns 0 (fourth argument) or pi (fifth argument), independent of 'x', for the upper and lower integration bounds of the second integration variable):

```
>>> def sinsin(x,y):
...     return numpy.sin(x)*numpy.sin(y)
...
>>> scipy.integrate.dblquad(sinsin,0,numpy.pi,lambda x: 0,lambda x: numpy.pi)
(3.9999999999999996, 4.4408920985006255e-14)
```

'quad' uses an adaptive Gaussian integration method (see your mathematics textbook on where this may be good or bad). Here is an example with fixed order

```
>>> scipy.integrate.fixed_quad(numpy.sin,0,numpy.pi, n=2)
(1.9358195746511373, None)
>>> scipy.integrate.fixed_quad(numpy.sin,0,numpy.pi, n=3)
(2.0013889136077436, None)
>>> scipy.integrate.fixed_quad(numpy.sin,0,numpy.pi, n=4)
(1.9999842284577225, None)
```

```
>>> scipy.integrate.fixed_quad(numpy.sin,0,numpy.pi, n=5)
(2.0000001102844709, None)
>>> scipy.integrate.fixed_quad(numpy.sin,0,numpy.pi, n=6)
(1.9999999994772708, None)
>>> scipy.integrate.fixed_quad(numpy.sin,0,numpy.pi, n=7)
(2.0000000000017879, None)
```

If you have data points already discretized on a grid, rather than functions, you can use these methods:

```
>>> x=numpy.arange(0,numpy.pi+1.e-8,numpy.pi/100)
>>> y=numpy.sin(x)
>>> numpy.trapz(x,y)
-1.9998355038874449
>>> x=numpy.arange(0,numpy.pi+1.e-8,numpy.pi/10)
>>> y=numpy.sin(x)
>>> numpy.trapz(x,y)
-1.9835235375094542
>>> x=numpy.arange(0,numpy.pi+1.e-8,numpy.pi/1000)
>>> y=numpy.sin(x)
>>> numpy.trapz(x,y)
-1.9999983550656633
>>> x=numpy.linspace(0,numpy.pi,101)
>>> y=numpy.sin(x)
>>> scipy.integrate.simps(x,y)
-2.0000159150483756
```

Romberg quadrature uses a trapezoidal integration on successively finer grids combined with an extrapolation to zero step size:

```
>>> x=numpy.linspace(0,numpy.pi,2**6+1)
>>> y=numpy.sin(x)
>>> scipy.integrate.romb(y,dx=x[1]-x[0],axis=0)
2.0000000000013216
```

5 1d Plotting in Matplotlib

```
>>> import matplotlib.pyplot as plt
>>> import numpy
>>> x=numpy.linspace(0,numpy.pi,20)
>>> y=numpy.sin(x)
>>> plt.xlabel("x")
>>> plt.ylabel("sin(x)")
>>> plt.plot(x,y, 'bo', label="sin(x)")
>>> plt.axis([-0.1, 3.2, 0, 1.1])
>>> plt.savefig("sin_curve.pdf", format="pdf")
```


Next step: add a figure title and a figure caption and plot the function with red lines on a fine grid

```
>>> plt.title("sin curve between zero and pi")
>>> x1=numpy.linspace(0,numpy.pi,2000)
>>> y1=numpy.sin(x1)
>>> plt.plot(x1,y1, 'r-', label="sin(x) smooth")
>>> plt.legend(loc="upper left")
>>> plt.savefig("sin_curve_2.pdf", format="pdf")
```

6 Interpolation

Add an interpolated function:

```
>>> f2=scipy.interpolate.interp1d(x,y,kind="linear")
>>> f3=scipy.interpolate.interp1d(x,y,kind="cubic")
>>> y2=f2(x1)
>>> y3=f3(x1)
>>> plt.plot(x1,y2, 'm-', label="sin(x) linear")
>>> plt.plot(x1,y2, 'm-', label="sin(x) cubic")
>>> plt.legend()
>>> plt.savefig("sin_curve_3.pdf", format="pdf")
```

Run a spline interpolation of a small data set:

```
>>> import numpy
>>> import scipy.interpolate as interpolate
>>> x=numpy.linspace(0,numpy.pi,4)
>>> y=numpy.sin(x)
>>> x1=numpy.linspace(0,numpy.pi,2000)
>>> tck = interpolate.splrep(x,y,s=0)
>>> y1=interpolate.splev(x1,tck,der=0)
```

Then plot it in matplotlib:

```
>>> plt.plot(x,y,'x',x1,y1,x1,numpy.sin(x1),x,y,'b')
>>> plt.legend(["Linear","Cubic","sin x"])
>>> plt.savefig("sin_curve_4.pdf", format="pdf")
```

7 Exercises

Practice, please! Make sure you understand the examples, repeat them, change them...