Python for Scientific Data Analysis

NumPy

Section 1: NumPy Arrays

NumPy, short for "Numerical Python", is one of the most important foundational packages for numerical computing in Python as McKinney says. In fact, it is so fundamental that it is very hard to do scientific Python without it. Even other workhorse packages that we will hear about -- e.g. Matplotlib, Pandas -- require it for their installation.

NumPy has some key powerful tools:

- *ndarray* - technically, native Python can support arrays but NumPy arrays are much faster. I.e. they allow fast fast array-oriented arithmetic operations and flexible broadcasting capabilities.
- Vectorized mathematical functions for fast operations on entire data arrays, obviating the need for loops.
- Tools for reading/writing array data to disk and working with memory-mapped files.
- Powerful linear algebra, random number generation, and Fourier transform capabilities.
- A C API for connecting NumPy with libraries written in C, C++, or FORTRAN. NumPy uses very fast C code under-the-hood.

McKinney gives a whole list of other things about why NumPy is so great (which it is). Now, NumPy doesn't do *everything* and there are other complementary packages that perform comparable tasks (esp. SciPy). And it doesn't really have built-in modeling capabilities like SciPy does. But MANY libraries depend on NumPy. So for Scientific Python, you need to know and master NumPy.

As mentioned before, to utilize NumPy's capabilities you need to import the library first. The standard is:

```
import numpy as np
```

Then different NumPy operations are simple: `np.[do_something]`.

Basics of NumPy Arrays

Simple Array Examples

The core of NumPy is the N-dimensional array object ("ndarray"). Or just an "***array***" (since NumPy arrays >> native Python arrays).

Here is a simple example of a NumPy array that we hard-code

```
import numpy as np

test_array=np.array([3,4,5,6])
test_array
#array([ 3,4,5,6])
```

And a simple example of a NumPy array that we compute:

```
import numpy as np

#generate some random data
test_array2 = np.random.randn(3,2)

test_array2
#array([[ 0.63736923,  1.11658941],
#       [-1.53912969, -0.10313421],
#       [-2.16932999, -1.0715993 ]])
```

Now, we can do different operations with this array. The simplest is arithmetic:

```
test_array2mult=test_array2*8
test_array2mult

#array([[  5.09895383,   8.93271532],
#       [-12.3130375 ,  -0.82507366],
#       [-17.35463994,  -8.57279442]])
```

Here, the multiplication by 8 is applied element-wise to the array `test_array2`.

Note that the multiplication is a *vectorized* multiplication, equivalent to the following in a C/Fortran-like nested loop:

```
test_array2mult=test_array2 #initialize data2, note: we will do this in a
 different way later

#note: we will worry about the shape attribute later
for i in range(test_array2.shape[0]):
 for j in range(test_array2.shape[1]):
  test_array2mult[i,j]=test_array2[i,j]*8

test_array2mult
#array([[  5.09895384,   8.93271528],
#       [-12.31303752,  -0.82507368],
#       [-17.35463992,  -8.5727944 ]])
```

Key Array Attributes

A NumPy array will have a lot of different attributes. But two key ones to keep in mind are: `.shape` and `.dtype`.

1. `.shape` -- refers to the dimensionality of the array. So for `test_array` in our example above, `test_array.shape` yields `(4,)`. That is, Python thinks that `test_array` is a one-dimensional array of length 4. In this case, in fact, `len(test_array)` and `test_array.shape` give the same answer!

   As is obvious, `test_array2` is NOT a 1D array. But it is a 3x2 array (e.g. like a matrix). Unsurpisingly, `test_array2.shape` yields `(3, 2)`. Similarly, the output array `test_array2mult` has the same dimensions.

2. `.dtype` -- refers to the type of data stored in the array. NumPy arrays -- unlike lists -- require a *fixed* data type. `test_array` looks like a bunch of integers, and `test_array.dtype` confirms: `dtype('int64')`.

   `test_array2`, though, looks like a collection of not-integers (e.g. floating point entries) and it is `test_array2.dtype` yields `dtype('float64')`. I.e. these are 64-bit floating numbers. By default, Python stores these numbers as double-precision. E.g. see this array:

```
a=np.array([np.e,np.pi])
a[0]
#2.718281828459045
a[1]
#3.141592653589793
print('{0:.50f}'.format(a[0]))
#2.71828182845904509079559829842764884233474731445312
print('{0:.55f}'.format(a[0]))
#2.7182818284590450907955982984276488423347473144531250000   #now we
start to get zeros
```

Note that by default Python is *displaying* 15 digits after the decimal point if you do not have a formatting prescribed. For almost all cases, double-precision is good enough. If you need more precision for whatever reason, 128 bit precision is the next step up, but this is beyond the scope of the course.

Creating NumPy Arrays From Lists and Tuples

The easiest way to create an array is to use the `array` function as we did in the two previous examples. E.g. `[something] = np.array[somethingelse]` in simple form. This function accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data.

In the first example, we created a NumPy array out of a *list* "[3,4,5,6]" (`np.array([3,4,5,6])`). Essentially this is a type conversion. We also could have done this: `mylist=[3,4,5,6]; myarray=np.array(mylist)` .

Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

```
mylist2=[[3,4,5],[6,7,8]]
myarray2=np.array(mylist2)
myarray2
#array([[3, 4, 5],
#       [6, 7, 8]])
myarray2.shape
#(2,3)
```

This also works for tuples:

```
mytuple=3,4,5,6,7,8
myarray3=np.array(mytuple)
#array([3, 4, 5, 6, 7, 8])

mytuple2=(3,4,5),(6,7,8)
myarray3b=np.array(mytuple2)
#array([[3, 4, 5],
#       [6, 7, 8]])
```

Altering Data Types in Arrays

Note that the default type for arrays. In the example above, it is 'int64' since all of the list entries are integers. In the `test_array2` example above, it is 'float64' (i.e. double precision floating point). Now we can convert the data type to something else.

E.g.

```
a=np.array([np.e,np.pi])
a.dtype #'float64'

a=np.array([np.e,np.pi],dtype='int64')
a.dtype #'int64'
a #array([2 3]) #note the conversion from floating point to integer!

a=np.array([np.e,np.pi],dtype='float128')
a.dtype #'float128'  #i.e. extended-precision floating point
```

Creating NumPy Arrays from Functions

Instead of only converting from lists/tuples, much like in other languages -- C, Fortran, IDL -- you can create arrays from nothing. There are a couple of key functions for generating arrays, their shapes, and what they do as listed below (largely copied from a concatenation of Fuhrer, Sect 4.5 and McKinney Sect 4.1):

Methods	Shape	Description
array((n,m))	(n,m)	convert input data to array; default is to copy
asarray((n,m))	(n,m)	convert to array; but do not copy if input is already an array
zeros((n,m))	(n,m)	Matrix filled with zeros
zeros_like(v)	(n,m)	Matrix filled with zeros whose dimensions (n,m) are equal to that of array *v*
ones((n,m))	(n,m)	Matrix filled with ones
ones_like(v)	(n,m)	Matrix filled with ones whose dimensions (n,m) are equal to that of array *v*
empty((n,m))	(n,m)	Create new array by allocating memory but do not populate w/ any values
empty_like(v)	(n,m)	zeros
full((n,m),*q*)	(n,m)	Matrix filled with *q*
ones_like(v)	(n,m)	Matrix filled with *q* whose dimensions (n,m) are equal to that of array *v*
diag((v,k))	(n,n)	(Sub-,super-)diagonal matrix along k-diagonal from vector *v* with dimensions (n,n)
eye(n,n), identity(n)	(n,n)	create square nxn identity matrix (1s on diagonal, 0s otherwise)
random.rand(n,m)	(n,m)	Matrix filled with uniformly distributed random numbers between 0 and 1
arange(n)	(n,)	First n integers
linspace(a,b,n)	(n,)	Vector with *n* equispaced points between *a* and *b*

Note that as before, you can set the data type as a keyword.

E.g.

```
a=np.full((50,50),np.pi)  #50 x50 matrix filled with pi
a=np.full((50,50),np.pi,dtype='int64') #50 x50 matrix filled with 3s due
to a type conversion
a=np.full((50,50),np.pi,dtype='float128')  #50 x50 matrix filled with ext
ended precision pi
```

Reading In Data and Writing Data with NumPy

At some point, you will want to read data in and write data to files with some Python package. Native Python has some functions for these -- e.g. the `.open`, `.write`, and `.close` functions. For very simple cases, these functions are fine but for formatted input and output -- especially *mixed format* -- you will want something a bit more powerful. Numpy's functions are a good option.

The full description of what NumPy can do is here: https://numpy.org/doc/stable/user/how-to-io.html

We are going to concentrate on two functions `loadtxt` and `savetxt` which load data from a file and save data to a file. Numpy loadtxt enables you to load numeric data that's stored in a text file.

So, for example, if you have row-and-column data that's stored in a text file, where the numbers are separated by commas (a so-called csv file), you can use Numpy loadtxt to load the data into your Python environment.

np.loadtxt

In the simplest case, `np.loadtxt` is called just with a filename: e.g. `np.loadtxt(filename)`. The full syntax looks like this:

```
np.loadtxt(fname, dtype='float', comments='#', delimiter=None, converters
=None, skiprows=0, usecols=None, unpack=False, ndmin=0)
```

There are too many examples to give, so I will do just one simple example and one complex one.

Simple example - we start with a file called `spectrum_oct17_adi.dat`, which is a file for the spectrum of the HIP 99770 b planet obtained with the Subaru Telescope. The columns are 1) wavelength (in microns), 2) flux density (mJy), 3) uncertainty on the flux density (mJy), and 4) signal-to-noise ratio of the detection.

The call is very simple:

```
import numpy as np
a=np.loadtxt('spectrum_oct17_adi.dat')
```

Now, if you print `a` you will see that this is an array of shape (22,4)

```
print(a)
#array([[ 1.1595614,  0.0501412,  0.047094 ,  1.06493  ],
       [ 1.1996971,  0.111398 ,  0.0346233,  3.22408  ],
       [ 1.2412219,  0.204607 ,  0.0313525,  6.57163  ],
       [ 1.284184 ,  0.227711 ,  0.0289869,  7.93797  ],
       [ 1.3286331,  0.151868 ,  0.0213903,  7.16115  ],
       [ 1.3746208,  0.0653105,  0.0301679,  2.16645  ],
       [ 1.4222002,  0.0671016,  0.0260064,  2.58312  ],
       [ 1.4714264,  0.137951 ,  0.0317177,  4.36435  ],
       [ 1.5223564,  0.13655  ,  0.028294 ,  4.84405  ],
       [ 1.5750496,  0.234941 ,  0.0258494,  9.21372  ],
       [ 1.6295663,  0.342276 ,  0.0277978, 12.6784   ],
       [ 1.6859701,  0.343458 ,  0.0344331, 10.1294   ],
       [ 1.744326 ,  0.261234 ,  0.0250239, 10.6481   ],
       [ 1.8047021,  0.194305 ,  0.0233974,  8.41     ],
       [ 1.8671678,  0.215942 ,  0.0296606,  7.34418  ],
       [ 1.9317957,  0.255128 ,  0.0394082,  6.52757  ],
       [ 1.9986603,  0.215387 ,  0.0262965,  8.31676  ],
       [ 2.0678396,  0.366825 ,  0.0334406, 11.285    ],
       [ 2.1394131,  0.40621  ,  0.029048 , 14.7394   ],
       [ 2.2134641,  0.374113 ,  0.0317572, 12.2824   ],
       [ 2.2900781,  0.302754 ,  0.0380343,  8.15881  ],
       [ 2.369344 ,  0.295742 ,  0.0551045,  5.49302  ]])

a.shape
(22,4)
```

From here, you can assign variables:

```
wvlh=a[:,0]
flux_mjy=a[:,1]
eflux_mjy=a[:,2]
snr=a[:,3]
```

Later, we will show how to plot these kinds of data with *Matplotlib*

Complex example - the above example was a simple read of file entries, all of which were

floating-point numbers. Now let's try something tricker ...

Here's a mixed-format file of GKM stars within 50 parsecs, where we record the name, the spectral type, and lots of other information about the stars, including their x-ray activity levels.

Now, `loadtxt` allows us to specify additional things: e.g. which columns to use (usecols) and the data types (dtypes). dtype can further be unpacked into 'names' and 'formats'. Watch what happens

```
infile='gkm_50pc_plus_gaia_and_cahk'
table=np.loadtxt(infile,dtype=np.unicode_)
hgca_name=table[:,0].astype(str)
hgca_spectype=table[:,1].astype(str)
hgca_bmv=table[:,10].astype(float)
hgca_cahk=table[:,11].astype(float)
hgca_mass=table[:,2].astype(float)
hgca_ra=table[:,4].astype(float)
hgca_dec=table[:,5].astype(float)
plx=table[:,6].astype(float)
```

Here's another example for the spectrum of AB Aur b:

```
dtypes={'names':('wavelength','flux','error'),\
'formats':(np.float64,np.float64,np.float64)}

x=np.loadtxt('spectrum_aloci_jan18.dat',usecols=range(3),dtype=dtypes)

wavelength=x['wavelength']
flux_a=x['flux']
error_a=x['error']
```

np.savetxt

Saving files follows the same format:

`np.savetxt(filename,(columns of data),other keywords)` EXCEPT THAT `savetxt` does this wonky thing where it saves arrays as *rows*, not columns. The work-around is to do some transposing with the `column_stack` function that we will hear about later. Much later we will also hear about an alternate write function within the AstroPy package that is, in my opinion, much better.

Finally, we can include formatted print output statements:

```
a=np.loadtxt('spectrum_oct17_adi.dat')
wvlh=a[:,0]
flux=a[:,1]
eflux=a[:,2]
np.savetxt('testme.txt',np.column_stack((wvlh,flux,eflux)),fmt="%.3f" " "
 "%.2f" " " "%.2f")

#1.160 0.05 0.05
1.200 0.11 0.03
1.241 0.20 0.03
1.284 0.23 0.03
1.329 0.15 0.02
1.375 0.07 0.03
1.422 0.07 0.03
1.471 0.14 0.03
1.522 0.14 0.03
1.575 0.23 0.03
1.630 0.34 0.03
1.686 0.34 0.03
1.744 0.26 0.03
1.805 0.19 0.02
1.867 0.22 0.03
1.932 0.26 0.04
1.999 0.22 0.03
2.068 0.37 0.03
2.139 0.41 0.03
2.213 0.37 0.03
2.290 0.30 0.04
2.369 0.30 0.06
```