

Astronomical data analysis using Python

Lecture 6

List Comprehension

```
In [61]: cities = ["Pune", "Mumbai", "Delhi", "Gauhati", "Chennai"]
newlist = []

for x in cities:
    if "u" in x:
        newlist.append(x)

print(newlist)
```

```
['Pune', 'Mumbai', 'Gauhati']
```

```
In [55]: cities = ["Pune", "Mumbai", "Delhi", "Gauhati", "Chennai"]

newlist = [x for x in cities if "u" in x]

print(newlist)
```

```
['Pune', 'Mumbai', 'Gauhati']
```

List comprehension syntax

```
newlist = [expression for item in iterable if condition == True]
```

The old list remains unchanged in the list comprehension. The iterable can be any iterable including a list. The if condition is optional.

```
In [60]: newlist = [x for x in range(10) if (x < 7) & (x > 3)]  
         print (newlist)
```

```
[4, 5, 6]
```

Nested lists

```
In [1]: k = [ (i,j) for i in range(3) for j in range(3) ]  
print (k)
```

```
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

List Comprehensions - Dos and Donts

For simple for-loops like we saw earlier, you **SHOULD** use list comprehensions. In fact, list comprehensions run faster than traditional loops.

But,

List comprehensions, especially nested ones, can obfuscate code. In such a case, it's better to use elaborate loops.

Installing third party modules

See the instructions at:

<https://docs.python.org/3/installing/index.html>
(<https://docs.python.org/3/installing/index.html>).

Please install numpy, scipy, matplotlib, astropy, astroquery modules on your computer. Except for astroquery all modules are also available in Google Colab. Even astroquery can be installed on Google Colab using `!pip install astroquery`

Python data processing using lists is very slow

Compared to compiled languages, for loops in Python run extremely slowly. The alternative is to define a more restrictive data structure than a list - **a numpy array**. Unlike lists, a numpy array contains data of only one type.

In exchange for this restriction, numpy arrays provide for unprecedented speed and convenience and make numerical computation practically possible in Python. The high speed of numpy is due to the fact that its underlying, highly optimised code is written in C.

All scientific and technical computing in Python today makes use of numpy in one way or another.

Adding two arrays - the slow Python way

```
In [2]: l1=[1,2,4]; l2=[4,8,6]  
print (l1+l2)
```

```
[1, 2, 4, 4, 8, 6]
```

```
In [3]: l3 = [0]*3  
for i in range(3):  
    l3[i] = l1[i] +l2[i]  
  
print (l3)
```

```
[5, 10, 10]
```


What we ideally want

We want `print (l1+l2)` to give us `[5,10,10]`

Such behaviour is common in Fortran 90, Matlab, IDL

Creating an n-dimensional array - numpy.ndarray

```
In [2]: import numpy as np
        print (np.version.version)

        a = np.array([1,2,3]) # Argument to np.array can be a list
        b = np.array((1,2,3)) # or a tuple
        print (type(a))
        print (type(b))
```

1.19.5

<class 'numpy.ndarray'>

<class 'numpy.ndarray'>

Adding Arrays with numpy

In [3]:

```
l1 = [1,2,3]
l2=[4,5,6]
a1 = np.array(l1)
a2 = np.array(l2)
print (l1+l2)
print (a1+a2)
```

```
[1, 2, 3, 4, 5, 6]
[5 7 9]
```

Arithmetic operations in numpy

```
In [4]: print (a1+a2) # addition  
        print (a1-a2) # subtraction  
        print (a1*a2) # multiplication  
        print (a1/a2) # division  
        print (a2%a1) # modulo  
        print (a1**2) # elementwise squared
```

```
[5 7 9]  
[-3 -3 -3]  
[ 4 10 18]  
[0.25 0.4 0.5 ]  
[0 1 0]  
[1 4 9]
```

Numpy ufuncs

add (+) log greater (>) subtract (-) log10 greater_equal (>=)

multiply (*) cos less (<) divide (/) arcos less_equal (<=)

remainder (%) sin logical_and absolute abs arcsin logical_or

floor tan logical_xor ceil arctan bitwise_and (&)

fmod cosh bitwise_or (|) conjugate sinh bitwise_xor (^)

minimum tanh bitwise_not (~) maximum sqrt rshift (>>)

power (**) equal (==) lshift (<<) exp not_equal (!=)

Why is an array not a list?

A numpy array is a table of elements (usually numbers), all of the **same** type.

Implicit and explicit datatype declaration

```
In [5]: import numpy as np
ai = np.array([1, 2, 3])
af = np.array([1, 2, 3.])
ac = np.array([1j, 2, 3.])
print (ai.dtype)
print (af.dtype)
print (ac.dtype)
af = np.array([1,2,3], float)
ac = np.array([1,2,3], complex)
afn = np.array([1,2], np.float) # numpy float type
```

```
int64
float64
complex128
```

Numpy float datatypes

- `np.float` (== Python float type)
- `np.float32` (single precision)
- `np.float64` (double precision)
- `np.double` (synonym, double prec.)
- `np.longdouble` (very platform dependent !!! `np.float96` or `np.float128`)

Other numpy datatypes

- Integer datatypes `np.int8` ... `np.int16` ... `np.int32` ... `np.int64`
- Unsigned (!) integer datatypes `np.uint8` ... `np.uint16` ... `np.uint32` ... `np.uint64`
- Complex datatypes `np.complex64` ... `np.complex128` ... `np.complex256`
- Boolean `np.bool`

Defining your own datatypes

```
In [4]: dt= np.dtype([('value',np.int),('status',np.bool)])  
a = np.array([(0,True),(100,False)],dtype=dt)  
print (a['value'])  
print (a['status'])  
print (a['value'][1])  
print (a[0]['status'])
```

```
[  0 100]  
[ True False]  
100  
True
```

Structured arrays are sometimes quite useful for reading binary data.

Convenience functions to create arrays

```
In [9]: z = np.zeros(3)
print (z)
o = np.ones(10)
print (o)
e = np.empty(10) # very fast, use only if array elements are guaranteed to be in
itialised
print (e) # uninitialised values, could be anything!(y)
print (np.zeros_like(e))
print (np.ones_like(e))
```

```
[0. 0. 0.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

Multidimensional Arrays

```
In [12]: l2 = [[1,2,3],[4,5,6]]  
         a2 = np.array(l2)  
         print (l2)  
         print (a2)
```

```
[[1, 2, 3], [4, 5, 6]]  
[[1 2 3]  
 [4 5 6]]
```

Swapping of axes

Note that, like C (and unlike FORTRAN), Python is 0-indexed and the indices have the slowest axis first and fast axis last, i.e. for a 2-D image, the fast axis (X-axis) which corresponds to the FITS NAXIS1 keyword, is the second index.

How best to handle this?

Accessing Arrays

```
In [12]: print l2[0][1]
          print a2[0][1]
          print a2[0,1] # works only with
          print a2[0,1:] # arrays !!!
          print a2[:,0] #
```

2

2

2

[2 3]

[1 4]

Determine size and shape

```
In [13]: # determine the dimensions  
# remember: l2=[[1,2,3],[4,5,6]]  
# how many rows, how many columns ?
```

```
print len(l2), len(l2[0])
```

```
print len(a2), len(a2[0])
```

```
print a2.shape # shape attribute
```

```
2 3
```

```
2 3
```

```
(2, 3)
```

Array attributes

- `array.ndim`: number of dimensions
- `array.shape`: the dimensions
- `array.size`: total number of elements
- `array.dtype`: type of the data

Initialising multidimensional arrays

```
In [16]: # np.zeros, np.ones, et al.  
# can be used to create multidim  
# arrays, too !  
  
o = np.ones( (3,10) )  
  
e = np.empty( (4,2,19), float)
```

Convenience functions

```
In [13]: b = np.linspace(-2.5,1.5,17)

# 17 equally spaced numbers from -2.5 to 1.5
# including the boundary values !

c = np.arange(-2.5,1.51,0.25)
# Unreliable because upperbound excluded, rounding errors, use linspace

x = np.linspace(0, 3.14, 100)

m = np.logspace(-2,3,9) # special for astronomers!

y = np.sin(x)
```

Slicing in numpy

```
In [16]: a = np.zeros( (4,4) )

print (a[2:4][2:4]) # does this work? no.

a[2:4,2:4] = 1 # Works
print (a)
a[0] = 2
print (a)
a[:,1:3] += 3
print (a)
# the array shape is unaltered by
# these operations
```

```
[]
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 1. 1.]
 [0. 0. 1. 1.]]
[[2. 2. 2. 2.]
 [0. 0. 0. 0.]
 [0. 0. 1. 1.]
 [0. 0. 1. 1.]]
[[2. 5. 5. 2.]
 [0. 3. 3. 0.]
 [0. 3. 4. 1.]
 [0. 3. 4. 1.]]
```

Rows and columns

```
In [17]: a = np.zeros( (4,4) )  
  
# accessing rows and columns  
  
a[0,:] # the first row  
  
a[0] = 4 # the first row set to 4  
  
# read a[i,...] as  
# a[i,<as many :, as needed>]  
# this is a specific numpy notation  
  
print (a[0,...])  
  
a[:,1] = 5 # the second column  
  
a[:,-1] = 6 # the last column
```

```
[4. 4. 4. 4.]
```

numpy ASCII file reading - obsoleted by astropy.io.ascii

```
fname = 'myfile.dat'
```

```
data = np.loadtxt(fname)
```

```
c24 = np.loadtxt(fname,usecols=(1,3)) # to select columns
```

```
n,w,x,y,z = np.loadtxt(fname,unpack = True)
```

```
w,y = np.loadtxt(fname,usecols=(1,3), unpack=True)
```

```
fname2 = 'mydata2.dat' # a different data file
```

```
data2 = np.loadtxt(fname2,delimiter='/', comments='!',skiprows=3)
```

default for comments is '#'

```
np.savetxt('result.dat', res,fmt='%d : \t %.6f')
```

Saving/Loading binary arrays

```
In [26]: data=np.arange(10)
data2 = np.arange(20)
np.savez('/tmp/myarrays',x=data,y=data2)

myarrays = np.load('/tmp/myarrays.npz')

print (myarrays['x'])

print (myarrays['y'])
```

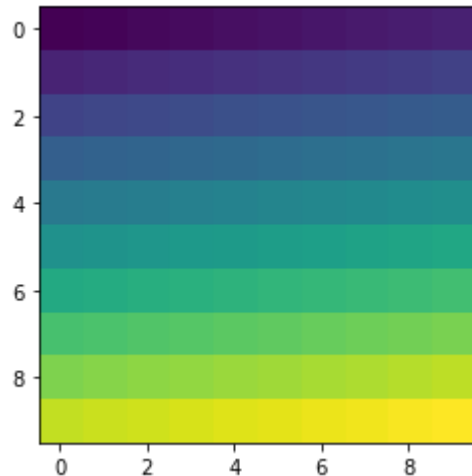
```
[0 1 2 3 4 5 6 7 8 9]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

more reliable than pickling files on disk.

Reshape and where

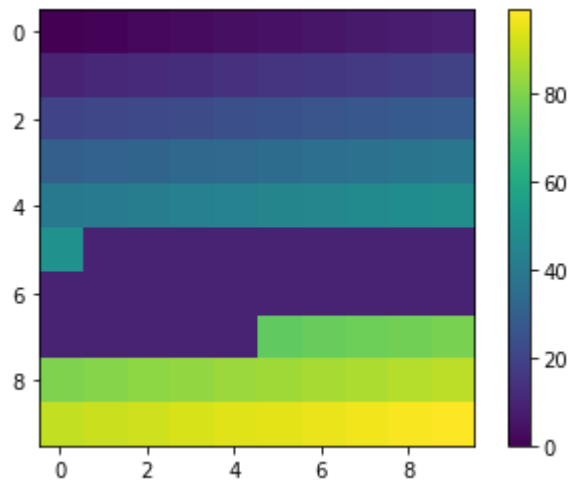
```
In [24]: import matplotlib.pyplot as plt  
x = np.arange(100).reshape((10,10))  
plt.imshow(x,interpolation='none')
```

Out[24]: <matplotlib.image.AxesImage at 0x7f8c3335cd30>



```
In [23]: x[np.where(np.logical_and(x > 50,x <75))] = 0  
plt.imshow(x,interpolation='none')  
plt.colorbar()
```

Out[23]: <matplotlib.colorbar.Colorbar at 0x7f8c3349b250>



np.where() may be the single most important function that you will use in numpy.

Numpy documentation with examples

There are over 400 functions in numpy (doing everything from rebinning to obtaining polynomial least squares fits) in numpy for array manipulation.

As time permits, please read through the excellent Numpy user guide at

<https://numpy.org/doc/stable/user/index.html>
(<https://numpy.org/doc/stable/user/index.html>).

We will cover only a fraction of the capabilities on numpy in this course.

Masked arrays - np.ma

Masked arrays are arrays that may have missing or invalid entries. Very common in astronomy.

```
In [5]: import numpy.ma as ma, numpy as np

x = np.array([1, 2, 3, -1, 5])
mx = ma.masked_array(x, mask=[0, 0, 0, 1, 0])
mx2 = ma.masked_array(x, mask=[1, 1, 0, 0, 0])
mx2 = ma.masked_array(x, mask=[True, True, False, False, False]) # equivalent more Pythonic syntax
print (mx2)
print (x.mean())
print (mx.mean())

mx/mx2 # Arithmetic operations on 2 masked arrays possible.

y = ma.masked_where(x>3,x)
print (y)

[-- -- 3 -1 5]
2.0
2.75
[1 2 3 -1 --]
```

Numpy functions and object methods

In [28]:

```
a = np.arange(100)

print (np.sum(a)) # Functional programming

print (a.sum()) # Object oriented programming

# The same underlying code provides both functional and OO calling functions.
```

4950

4950

Histograms

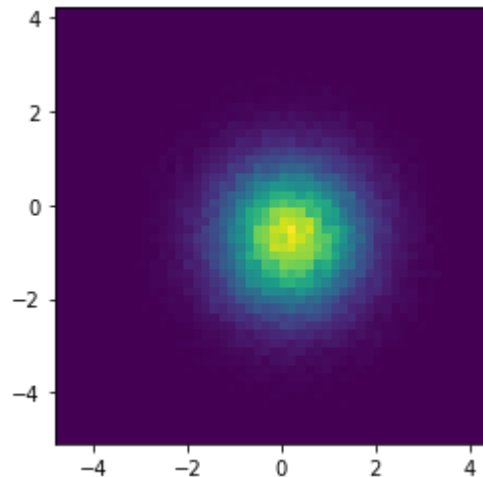
```
In [30]: x, y = np.random.randn(2, 100000)

H, xedges, yedges = np.histogram2d(x, y, bins=50)

extent = [xedges[0], xedges[-1], yedges[0], yedges[-1]]

plt.imshow(H, extent=extent)
```

```
Out[30]: <matplotlib.image.AxesImage at 0x7f8c33335df0>
```



Matrix objects

```
In [52]: x1 = np.array([1, 2, 3], float)
          x2 = np.matrix(x1)
          A = np.identity(3)
          A = 5* np.mat(A)
          B = x2*A # matrix multiplication
          np.linalg.inv(A)
```

```
Out[52]: matrix([[0.2, 0. , 0. ],
                 [0. , 0.2, 0. ],
                 [0. , 0. , 0.2]])
```

* represents matrix-matrix matrix-vector or vector-matrix multiplication.

Numpy capabilities

- Discrete Fourier Transform (`np.fft`)
- Linear algebra (`np.linalg`): cholesky, svd, eigenvalues/vectors, matrix inversion, solving equations (LAPACK)
- Random sampling (`np.random`)
- statistics: covariance, correlation, variances, histograms
- polynomials: root finding, fitting data
- Matrix library (`np.matlib`)
- **f2py** - a FORTRAN to Python interface generator

How fast is numpy?

- 100 times faster than with a for loop with native Python types
- Somewhat slower than a modern FORTRAN/C++ compiler with -O3 optimisations
- comparable speed to Matlab

But this is run speed. In development and debugging time, Python is far ahead of FORTRAN/C++. But if **speed is really critical** to your code, you can profile your Python code and rewrite slower parts in C/C++/FORTRAN.