Python Advance Course via Astronomy street

Sérgio Sousa (IA)
ExoEarths Team (http://www.astro.up.pt/exoearths/)

Python Advance Course via Astronomy street

Advance Course Outline:

- Lesson 1: Python basics (T + P)
- Lesson 2: Python with Numpy (T + P)
- Lesson 3: Matplotlib, Science and Astronomy modules

Python Advance Course via Astronomy street

Lesson 2: Python with Numpy and Matplotlib

- Object Oriented (OO) – Definition of objects/classes
- Numpy: creating and manipulation numerical data

python

Object Oriented Paradigm

Why shall we care about Object Oriented programing???

- Objects are a metaphor that permits a good division of the code and abstraction of the reality
- Allow an easy way to implement changes to existing code or to add new code without changing/destroying much of what was done before.
- Python is based on OO and so are most of the libraries available.
- Almost everything in Python is an object (even functions are in fact Python objects)

Defining Objects/Classes

Simple approach

Class Variables

Instance Variables

Instance Methods (self)

Class Methods (static)

```
class Star:
  """ A simple Star class"""

  grav = 6.67384e-11 # Gravitional constant

  def __init__(self, massin,radiusin):
    self.mass = massin
    self.radius = radiusin

  def get_surface_gravity(self):
    return Star.grav*self.mass/self.radius**2

  @staticmethod
  def get_spectral_types():
    return ['O','B','A','F','G','K','M']


## My functions:


### Main program:
def main():
  print "Hello"
  s1=Star(1.1,0.98)
  print s1.mass
  print s1.radius
  print s1.get_surface_gravity()
  print Star.get_spectral_types()
```

Running this code:

```
Hello
1.1
0.98
7.64392336526e-11
['O', 'B', 'A', 'F', 'G', 'K', 'M']
```

python

OO things to explore

Object built-in functions and Operators Overloading

1. __init__ (self [,args...])

Constructor (with any optional arguments)
Sample Call : obj = className(args)

2. __del__(self)

Destructor, deletes an object. Sample Call : dell obj

3. __repr__(self)

Evaluatable string representation. Sample Call : repr(obj)

4. __str__(self)

Printable string representation. Sample Call : str(obj)

5. __cmp__ (self, x)

Object comparison. Sample Call: cmp(obj, x)

6. __add__(self, other)

Adding objects: Overloading of + operator. Sample Call: obj1 + ob2

ObjectOverloading (~/Dropbox/Python/AdvancedCourse/examples) - gedit

```python
#!/usr/bin/python

class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print v1 + v2
```

```
$ ./ObjectOverloading.py
Vector (7, 8)
```

OO things to explore

OO Keywords to read about:

- **Abstraction:** The extraction of the key carachteristics of a concept or entity from the real world to allow their representation in a computer.
- **Encapsulation:** The values of the variables inside an object are private, unless methods are written to pass that information outside of the object. (Not practicable in Python)
- **Inherance:** Each subclass inherits all variables and methods of its superclass. (e.g. Animal is a superclass of Dog and Cat
- **Polymorphism:** Each object can behave as a super class object (e.g. Cat and Dog can behave as Animal)

Other interesting OO keywords to explore for complex applications:

  - Abstract Classes
  - Interfaces

Good Practices in Python

- Explicit variable names (no need of a comment to explain what is in the variable)
- Style: spaces after commas, around =, etc. A certain number of rules for writing “beautiful” code (and, more importantly, using the same conventions as everybody else!) are given in the Style Guide for Python Code and the Docstring Conventions page (to manage help strings).
- Except some rare cases, variable names and comments in English.
- Good identation (forced in Python)

Python Advance Course via Astronomy street

Lesson 2: Python with Numpy and Matplotlib

- Object Oriented (OO) – Definition of objects/classes
- Numpy: creating and manipulation numerical data

python

Numpy

```
>>> import numpy
>>> help(numpy)
```

```
sousasag@asusg51jx: ~/posdoc/docs/isep/AULAS/PPROG
Help on package numpy:

NAME
    numpy

FILE
    /usr/lib/python2.7/dist-packages/numpy/__init__.py

DESCRIPTION
    NumPy
    =====

    Provides
      1. An array object of arbitrary homogeneous items
      2. Fast mathematical operations over arrays
      3. Linear Algebra, Fourier Transforms, Random Number Generation
```

Numpy Array vs. Python list

example_timeit_numpy.py

```python
#!/usr/bin/python
## My first python code

##imports:
import numpy as np
import timeit

## My functions:

def normal_list():
  L = range(1000000)
  L2 = [i**2 for i in L]


def numpy_array():
  a = np.arange(1000000)
  a2 = a**2

### Main program:
def main():
  print "Normal List:"
  timer = timeit.Timer("normal_list()", setup="from __main__ import normal_list")
  print timer.timeit(1)*1000. , 'mili seconds'
  print "Numpy Array:"
  timer = timeit.Timer("numpy_array()", setup="from __main__ import numpy_array")
  print timer.timeit(1)*1000. , 'mili seconds'


if __name__ == "__main__":
    main()
```

```
$ ./example_timeit_numpy.py
Normal List:
140.730142593 mili seconds
Numpy Array:
7.87591934204 mili seconds
```

Things that you can do with Numpy

- Create Arrays: direct definition

```
>>> import numpy as np
>>> a = np.array([0,1,2,3])  # 1D array
>>> b = np.array([[0, 1, 2], [3, 4, 5]]) # 2D array: 2 x 3 array
>>> c = np.array([[[1], [2]], [[3], [4]]]) # 3D Array
>>> # evenly spaced:
...
>>> d = np.arange(10)
>>> d
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> e = np.arange(1,9,2) # start, end (exlusive), step
>>> e
array([1, 3, 5, 7])
>>> # by a number of points
...
>>> f = np.linspace(0, 1, 6) # start, end, num-points
>>> f
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
>>>
>>> a
array([0, 1, 2, 3])
>>> a.ndim
1
>>> b
array([[0, 1, 2],
       [3, 4, 5]])
>>> b.ndim
2
>>> len(a)
4
>>> b.shape
(2, 3)
```

Same as range in built in Python

Defining a linear space array

Some properties of the array object

This is a tuple

Things that you can do with Numpy

- Create Arrays: automatic definitions

```
>>> # creation of predefined arrays:
...
>>> a = np.ones((3, 3))
>>> # reminder: (3, 3) is a tuple
... a
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>>
>>> b = np.zeros((2,2))
>>> b
array([[ 0.,  0.],
       [ 0.,  0.]])
>>>
>>> c = np.eye(3)
>>> c
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>>
>>> d = np.diag([1,2,3,4])
>>> d
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

ones and zeros also work for 1D

Defining an Identity matrix

Defining a diagonal matrix

Things that you can do with Numpy

- Create Arrays: setting different data types

```
>>> #creating with different data types
...
>>> a = np.array([1, 2, 3])
>>> a.dtype
dtype('int64')
>>>
>>> a = np.array([1., 2., 3.])
>>> a.dtype
dtype('float64')
>>>
>>> a = np.array([1, 2, 3], dtype=float) #bool, complex
>>> a.dtype
dtype('float64')
>>> a = np.array([1, 2, 3], dtype=complex) #bool, complex
>>> a.dtype
dtype('complex128')
```

automatic definition

Forcing the types

Things that you can do with Numpy

- Indexing in Arrays – getting the values that you need

```
>>> #indexing and Slicing
...
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[0], a[2], a[-1]
(0, 2, 9)
>>>
>>> #index in numpy 2D: 2ways:
... a = np.diag(np.arange(3))
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 2]])
>>> a[1][1]
1
>>> a[1,1]      #using a tuple
1
>>> a[(1,1)]
1
>>> a[1,2] = 26
>>> a
array([[ 0,  0,  0],
       [ 0,  1, 26],
       [ 0,  0,  2]])
>>> a[1]# geting a line
array([ 0,  1, 26])
>>> a[:,2]# getting a row
array([ 0, 26,  2])
```

Indexing arrays is the same as for lists

You can also use tuples for arrays,
simpler notation for n>1 Dimensional arrays

Getting lines and/or rows from a matrix

Things that you can do with Numpy

- Create Arrays: Slicing and Views

```
>>> #slicing:
... a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3] # start;end;step
array([2, 5, 8])
>>> a[:4]
array([0, 1, 2, 3])
>>> a[2:]
array([2, 3, 4, 5, 6, 7, 8, 9])
>>> a[::2]
array([0, 2, 4, 6, 8])
```

Sintaxe for slicing:
array[start:end:step]

Some useful variations

```
>>> #slicing create views, using the same memory... Careful
...
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a[::2]
>>> b
array([0, 2, 4, 6, 8])
>>> b[0] = 26
>>> b
array([26,  2,  4,  6,  8])
>>> a
array([26,  1,  2,  3,  4,  5,  6,  7,  8,  9])
>>> # to copy use:
... b = a[::2].copy() # this force a copy
>>> b[0] = 12
>>> b
array([12,  2,  4,  6,  8])
>>> a
array([26,  1,  2,  3,  4,  5,  6,  7,  8,  9])
```

Important to be aware:

Slicing create views that share the same memory of the original array;

If you want to copy you need to force the copy;

Things that you can do with Numpy

- Create Arrays: Applying masks (can be used as where in IDL)

```
>>> #masks (similar to where in IDL)
...
>>> a = np.random.random_integers(0, 20, 15)
>>> a
array([ 6, 18,  6,  3,  5, 14, 15, 16,  8,  1, 20, 20,  5,  7,  8])
>>> mask = (a % 2 == 0)
>>> mask
array([ True,  True,  True, False, False,  True, False,  True,  True,
       False,  True,  True, False, False,  True], dtype=bool)
>>> even_a = a[mask] # this create a copy, not a view
>>> even_a
array([ 6, 18,  6, 14, 16,  8, 20, 20,  8])
>>>
>>> # useful for taging
... a[a % 2 == 0] = -1
>>> a
array([-1, -1, -1,  3,  5, -1, 15, -1, -1,  1, -1, -1,  5,  7, -1])
>>>
>>> a[[0,1,3]] = 26 # use a list to change n values in one line
>>> a
array([26, 26, -1, 26,  5, -1, 15, -1, -1,  1, -1, -1,  5,  7, -1])
```

Next slide for random numbers (wait for it...)

Defining a mask (which will be an array of booleans)

Applying the mask

Nice for tagging (check example)

Can also be used to assign a value to different places at once

Things that you can do with Numpy

- Create Arrays: Applying masks (can be used as where in IDL)

  Using np.where()

Define some array with numbers

```
>>> import numpy as np
>>> x = np.arange(15)+12
>>> x
array([12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26])
>>> ix = np.where(x % 2 == 0)
>>> ix
(array([ 0,  2,  4,  6,  8, 10, 12, 14]),)
>>> x2 = x[ix]
>>> x2
array([12, 14, 16, 18, 20, 22, 24, 26])
>>> np.where(x % 2 == 0, x, -1)
array([12, -1, 14, -1, 16, -1, 18, -1, 20, -1, 22, -1, 24, -1, 26])
>>> x
array([12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26])
>>> x = np.where(x % 2 == 0, x, -1)
>>> x
array([12, -1, 14, -1, 16, -1, 18, -1, 20, -1, 22, -1, 24, -1, 26])
>>>
```

Using where.
Similar as a mask

You can also tag with where

Can also be used to assign a value to different places at once

Things that you can do with Numpy

- Create Arrays: Random numbers generation

```
>>> # creation of random numbers:
...
>>> a = np.random.rand(4) # uniform in [0, 1]
>>> a
array([ 0.292304  ,  0.66154467,  0.88626098,  0.67117646])
>>>
>>> b = np.random.randn(4) # Gaussian
>>> b
array([-1.07921219, -0.62811542,  0.29934704,  0.12727723])
>>>
>>> np.random.seed(1234) # Setting the random seed
>>>
>>> c = np.random.random_integers(1,50,5) # 5 random integers
>>> c
array([48, 20, 39, 13, 25])
>>>
>>> s = np.random.poisson(5, 10) # see help(np.random) for more distributions
>>> s
array([ 5,  1,  4,  2,  6,  6,  6, 10,  2,  3])
```

The standard random common to many languages

A gaussian random number generation. G(0,1)

Important: Careful with the seed to generate the random numbers

To generate the next Euromillions numbers

You can also obtain random numbers following many other statistical distributions. ex. poisson

Things that you can do with Numpy

- Create Arrays: Random numbers generation – The statistical distributions available

```
==================== ==========================================================
Univariate distributions
==============================================================================
beta                 Beta distribution over ``[0, 1]``.
binomial             Binomial distribution.
chisquare            :math:`\chi^2` distribution.
exponential          Exponential distribution.
f                    F (Fisher-Snedecor) distribution.
gamma                Gamma distribution.
geometric            Geometric distribution.
gumbel               Gumbel distribution.
hypergeometric       Hypergeometric distribution.
laplace              Laplace distribution.
logistic             Logistic distribution.
lognormal            Log-normal distribution.
logseries            Logarithmic series distribution.
negative_binomial    Negative binomial distribution.
noncentral_chisquare Non-central chi-square distribution.
noncentral_f         Non-central F distribution.
normal               Normal / Gaussian distribution.
pareto               Pareto distribution.
poisson              Poisson distribution.
power                Power distribution.
rayleigh             Rayleigh distribution.
triangular           Triangular distribution.
uniform              Uniform distribution.
vonmises             Von Mises circular distribution.
wald                 Wald (inverse Gaussian) distribution.
weibull              Weibull distribution.
zipf                 Zipf's distribution over ranked data.
==================== ==========================================================
```

```
>>> help(np.random)
```

Things that you can do with Numpy

- Arrays: Numerical Operations

```
>>> a = np.array([1,2,3,4])
>>> a + 1
array([2, 3, 4, 5])
>>> 2**a
array([ 2,  4,  8, 16])
>>>
>>> # arrays are objects, operators were overloaded
... b = np.ones(4) + 1
>>> b
array([ 2.,  2.,  2.,  2.])
>>> a - b
array([-1.,  0.,  1.,  2.])
>>> a * b
array([ 2.,  4.,  6.,  8.])
>>>
>>> # warning: NOT matrix multiplication
...
>>> c = np.ones((3,3))
>>> c
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> c * c
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>>
>>> # to multiply matrices:
...
>>> c.dot(c)
array([[ 3.,  3.,  3.],
       [ 3.,  3.,  3.],
       [ 3.,  3.,  3.]])
```

Numpy arrays are objects.

Most of the operators where overloaded to represent what we would like.

(One of the advantages of OO programing)

We can do straightforward operations with arrays (similar to IDL)

Careful with the multiplication of arrays:

Use array.dot(array) to do a proper algebra multiplication of arrays

Things that you can do with Numpy

- Create Arrays: comparisons and reductions

```
>>> # comparisons
...
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> a == b
array([False,  True, False,  True], dtype=bool)
>>> a > b
array([False, False,  True, False], dtype=bool)
>>> #logical operations
... np.all([True, True, False])
False
>>> np.any([True, True, False])
True
```

The comparisons operatores are also overloaded. You can use it to compare directly arrays.

In addition you also have some useful logical operations in array

There are several reductions that you can do directly from the arrays.
- sum all elements;
- get the mean;
- standard deviation;
- median (is a class method);
- get extremes in array;
- get index of extremes;

Another useful method is the sort(). The default algorithm is the quicksort, but you can choose others. (check with >>>help(np.sort))

```
>>> # some basic reductions:
...
>>> x = np.array([3, 10, 5, 1, 2, 7, 4])
>>> np.sum(x)
32
>>> x.sum()
32
>>> x.mean()
4.5714285714285712
>>> x.std()
2.8713930346059686
>>> np.median(x) # there is no x.median()...
4.0
>>> x.min()
1
>>> x.max()
10
>>> x.argmin() # index of minimum
3
>>> x.argmax() # index of maximum
1
>>> x.sort()
>>> x
array([ 1,  2,  3,  4,  5,  7, 10])
```

Things that you can do with Numpy

- Create Arrays: reductions in 2D

```
>>> #notes on 2D or higher
...
>>> x = np.array([[1, 1], [2, 2]])
>>> x
array([[1, 1],
       [2, 2]])
>>>
>>> x.sum()
6
>>>
>>> x.sum(axis=0)
array([3, 3])
>>>
>>> x.sum(axis=1)
array([2, 4])
>>>
>>> x[:, 0].sum()
3
>>>
>>> x.T # transpose
array([[1, 2],
       [1, 2]])
```

When you have 2D arrays (or with more dimensions) you can also use the methods.

You can also select the axis that you want to reduce using the option inside the method.

Or alternatively you use what you learn on the indexing/slicing

<u>Special note:</u>
How to quickly transpose a matrix

Things that you can do with Numpy

- Create Arrays: data types casting and rounding

```
>>> # data types and Casting
... np.array([1, 2, 3]) + 1.5 # bigger type win
array([ 2.5,  3.5,  4.5])
>>>
>>> # assignment don't change type
... a = np.array([1, 2, 3])
>>> a.dtype
dtype('int64')
>>> a[0] = 1.9 # float will be truncated
>>> a
array([1, 2, 3])
>>>
>>> #forced cast:
... a = np.array([1.7, 1.2, 1.6])
>>> b = a.astype(int) # <-- truncates to integer
>>> b
array([1, 1, 1])
>>>
>>> # Rounding
...
>>> a = np.array([1.2, 1.5, 1.6, 2.5, 3.5, 4.5])
>>> b = np.around(a)
>>> b
array([ 1.,  2.,  2.,  2.,  4.,  4.])
>>> c = np.around(a).astype(int)
>>> c
array([1, 2, 2, 2, 4, 4])
```

Casting: is the transformation of a variable between different data types

When making operations with different data types, the output will be the bigger data type involved.

Careful with Down - casting
Example: Floats will be truncated when passing to integers.

You can also force a cast of an array.

You can use the around() to round values.

The result is casted, and you may want to force the cast to a different data-type.

Things that you can do with Numpy

- Polynomials

```
>>> #polynomials
... #3x**2 + 2*x - 1:
... p = np.poly1d([3, 2, -1])
>>> p(0)
-1
>>> p.roots
array([-1.        ,  0.33333333])
>>> p.order
2
>>> p.c
array([ 3,  2, -1])
>>>
>>> q = p * p
>>> q
poly1d([ 9, 12, -2, -4,  1])
>>> q.order
4
```

Polynomials with 1 variable can also be represented by a numpy object (poly1d).

Definition of a polynomial can be done using its coeficients in a list:
Example: 3x^2 + 2x – 1:
List: [3,2,-1] → [a2,a1,a0]

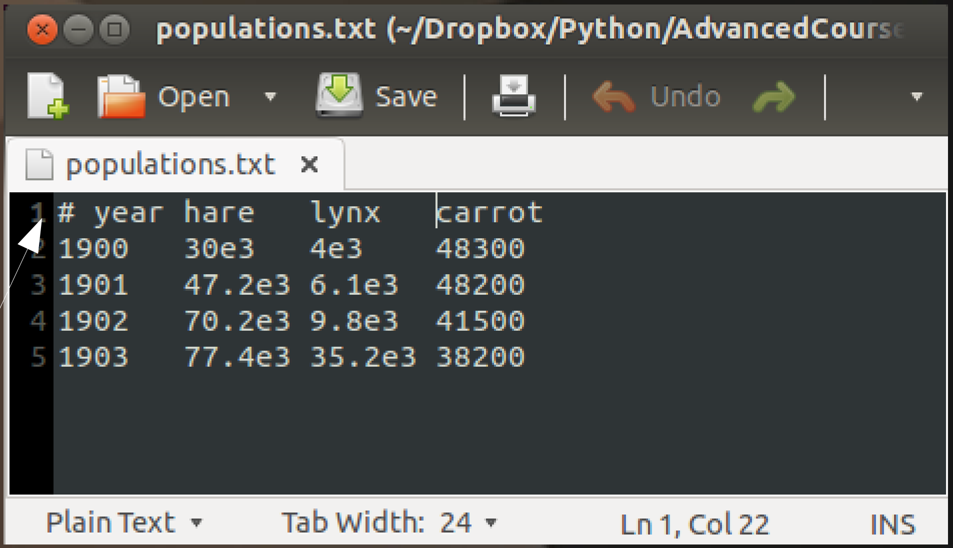
There are some useful methods:
- getting the roots
- getting the order
- getting the coeficients

You can also perform direct operations with polynomials.

Things that you can do with Numpy

- Reading and writing data to files with Numpy

```
>>> import numpy as np
>>> data = np.loadtxt('populations.txt')
>>> data
array([[  1900.,  30000.,   4000.,  48300.],
       [  1901.,  47200.,   6100.,  48200.],
       [  1902.,  70200.,   9800.,  41500.],
       [  1903.,  77400.,  35200.,  38200.]])
```



Using loadtxt() you can directly obtain the data.

Careful with header in files. They should start with the spetial caracther '#'

You can also save the data directly to a file with savetxt().

```
>>> np.savetxt('pop2.txt', data)
>>>
sousasag@asusg51jx:~/Dropbox/Python/AdvancedCourse/examples$ more pop2.txt
1.900000000000000000e+03 3.000000000000000000e+04 4.000000000000000000e+03 4.830000000000000000e+04
1.901000000000000000e+03 4.720000000000000000e+04 6.100000000000000000e+03 4.820000000000000000e+04
1.902000000000000000e+03 7.020000000000000000e+04 9.800000000000000000e+03 4.150000000000000000e+04
1.903000000000000000e+03 7.740000000000000000e+04 3.520000000000000000e+04 3.820000000000000000e+04
```