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 and Matplotlib (T + P)
- Lesson 3: Science and Astronomy modules (Scipy, Pyfits, Pyraf) (T + P)

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
- Plotting with Matplotlib

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

```
#!/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
- Plotting with Matplotlib

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()

```
>>> 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])
>>>
```

Define some array with numbers

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;

```
>>> # 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])
```

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

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]

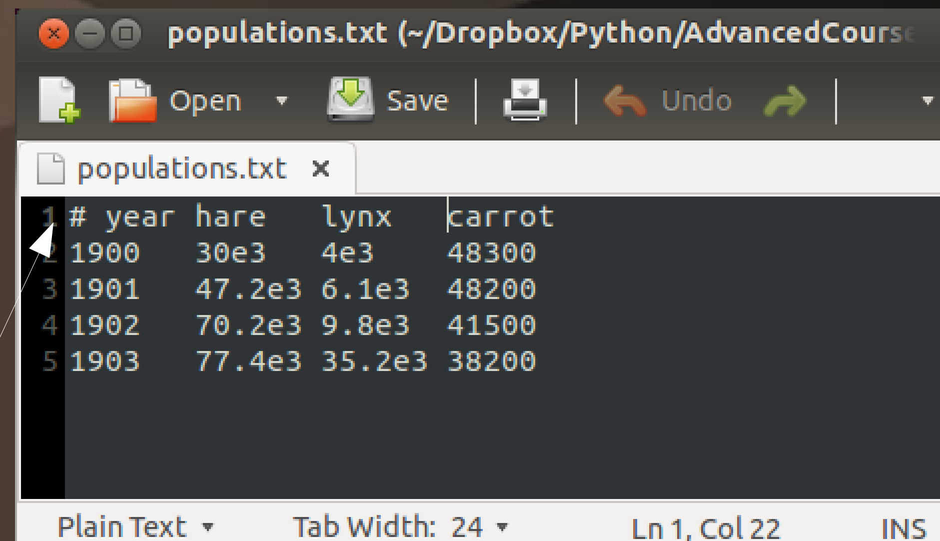
There are some useful methods:
- getting the roots
- getting the order
- getting the coeficients

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
```

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
- Plotting with Matplotlib

python

Matplotlib

```
>>> import matplotlib
>>> help(matplotlib)
```

```
Help on package matplotlib:

NAME
    matplotlib - This is an object-oriented plotting library.

FILE
    /usr/lib/pymodules/python2.7/matplotlib/__init__.py

DESCRIPTION
    A procedural interface is provided by the companion pyplot module,
    which may be imported directly, e.g::

        from matplotlib.pyplot import *

    To include numpy functions too, use::

        from pylab import *

    or using ipython::

        ipython -pylab

    For the most part, direct use of the object-oriented library is
    encouraged when programming; pyplot is primarily for working
    interactively.  The
    exceptions are the pyplot commands :func:`~matplotlib.pyplot.figure`,
    :func:`~matplotlib.pyplot.subplot`,
    :func:`~matplotlib.pyplot.subplots`,
```

Using Matplotlib

Simple plot – using procedural interface (pyplot)

```
>>> import numpy as np  #It will be useful to deal with data
>>> import matplotlib.pyplot as pl  #procedural interface
>>>
>>> #simple plot
...
>>> X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
>>> C, S = np.cos(X), np.sin(X)
>>> pl.plot(X, C)
[<matplotlib.lines.Line2D object at 0x2c4f410>]
>>> pl.plot(X, S)
[<matplotlib.lines.Line2D object at 0x350d490>]
>>>
>>> pl.show()
```

numpy useful to deal with data arrays

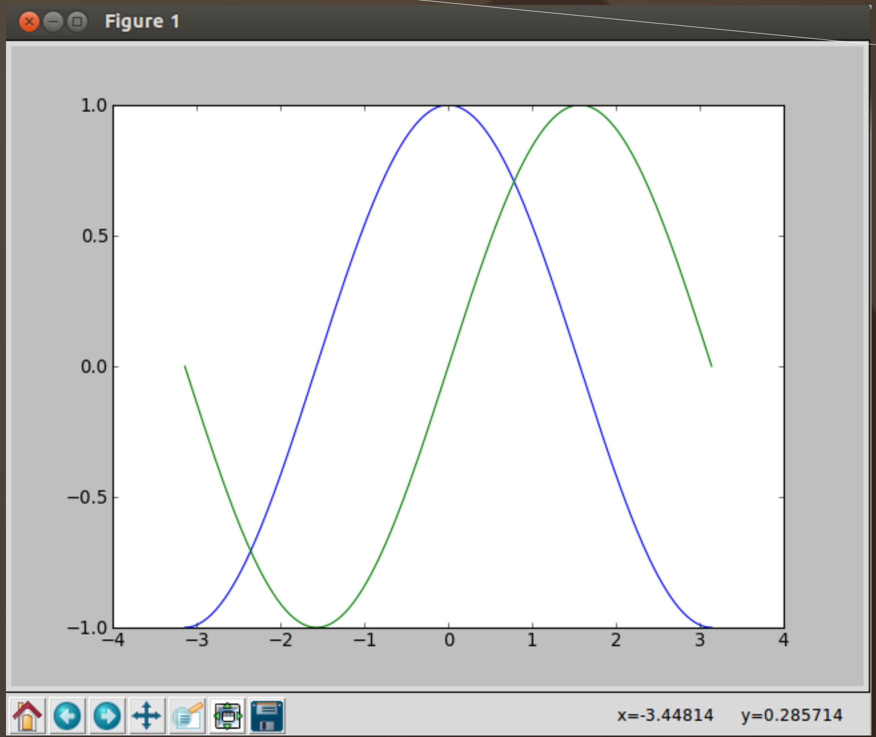
Pyplot – the module to “ignore” objects

Creation of data (x, cos(x)) and (x, sin(x))

Plot each set of data

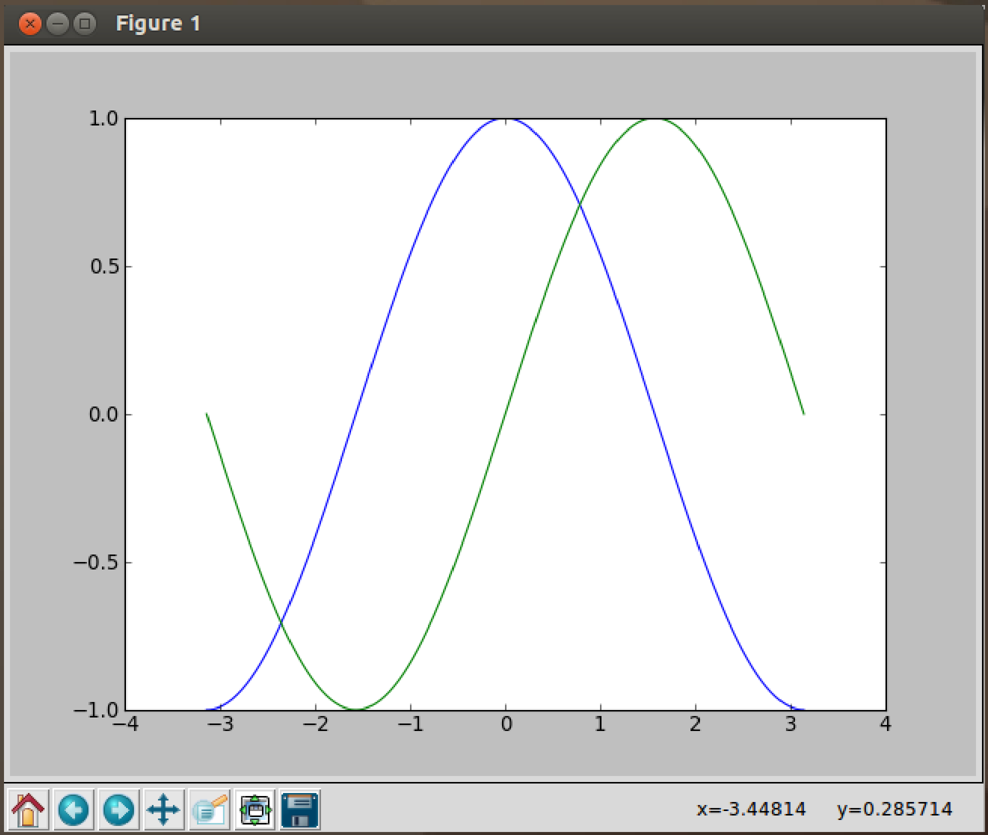
– note that objects are still created and in memory...

Show comand to open the plot window (Freezing the Python interpreter)

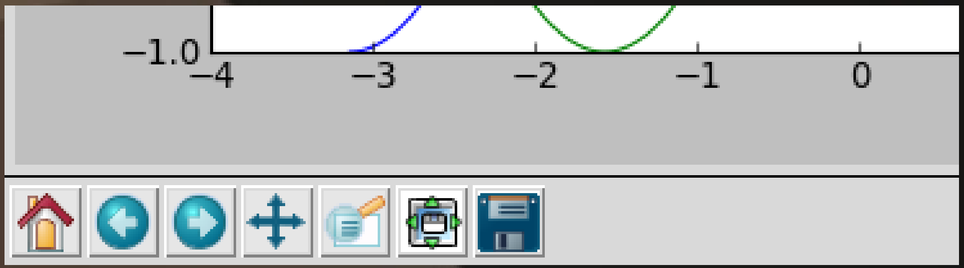


Using Matplotlib

The Plotting window - Figure



The plot window has some nice interactive features that you can easily explore



 Show original plot

 Undo/Redo visualization

 Navigation in the plot

 Zoom Rectangle

 Customize Subplots

 Saving/Exporting Figure

Using Matplotlib

Simple plot – changing default settings – procedural interface

```
# changing some default parameters

# Create a figure of size 8x6 points, 80 dots per inch
pl.figure(figsize=(8, 6), dpi=80)

# Create a new subplot from a grid of 1x1
pl.subplot(1, 1, 1)

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)

# Plot cosine with a blue continuous line of width 1 (pixels)
pl.plot(X, C, color="blue", linewidth=1.0, linestyle="-")

# Plot sine with a green continuous line of width 1 (pixels)
pl.plot(X, S, color="green", linewidth=1.0, linestyle="-")

# Set x limits
pl.xlim(-4.0, 4.0)

# Set x ticks
pl.xticks(np.linspace(-4, 4, 9, endpoint=True))

# Set y limits
pl.ylim(-1.0, 1.0)

# Set y ticks
pl.yticks(np.linspace(-1, 1, 5, endpoint=True))

# Save figure using 72 dots per inch
# savefig("exercice_2.png", dpi=72)

# Show result on screen
pl.show()
```

Defining the figure

Defining a subplot of the figure

Creation of data
(x, cos(x)) and (x, sin(x))

Plot data, setting color, linewidth and linestyle

Defining x axis limits

Difining x axis thicks

Defining y axis limits

Difining y axis thicks

Possibility to save Figure as .png file

Show comand to open the plot window
(Freezing the Python interpreter)

Using Matplotlib

Ploting lines – linestyles and markers :: >>> help(pl.plot)

```
The following format string characters are accepted to control
the line style or marker:

================    ===============================
character           description
================    ===============================
``'-'``             solid line style
``'--'``            dashed line style
``'-.'``            dash-dot line style
``':'``             dotted line style
``'.'``             point marker
``','``             pixel marker
``'o'``             circle marker
``'v'``             triangle_down marker
``'^'``             triangle_up marker
``'<'``             triangle_left marker
``'>'``             triangle_right marker
``'1'``             tri_down marker
``'2'``             tri_up marker
``'3'``             tri_left marker
``'4'``             tri_right marker
``'s'``             square marker
``'p'``             pentagon marker
``'*'``             star marker
``'h'``             hexagon1 marker
``'H'``             hexagon2 marker
``'+'``             plus marker
``'x'``             x marker
``'D'``             diamond marker
``'d'``             thin_diamond marker
``'|'``             vline marker
``'_'``             hline marker
================    ===============================
```

Using Matplotlib

Ploting lines – colors :: >>> help(pl.plot)

```
The following color abbreviations are supported:

==========  ========
character   color
==========  ========
'b'         blue
'g'         green
'r'         red
'c'         cyan
'm'         magenta
'y'         yellow
'k'         black
'w'         white
==========  ========

In addition, you can specify colors in many weird and
wonderful ways, including full names (``'green'``), hex
strings (``'#008000'``), RGB or RGBA tuples (``(0,1,0,1)``) or
grayscale intensities as a string (``'0.8'``).  Of these, the
string specifications can be used in place of a ``fmt`` group,
but the tuple forms can be used only as ``kwargs``.
```

Example of using RGB to get a blue color:

```
>>> pl.plot(X, C, color="#0000CC", linewidth=3.0, linestyle="-.")
[<matplotlib.lines.Line2D object at 0x3537350>]
>>> pl.show()
```

Using Matplotlib

Simple plot with legend – using procedural interface (pyplot)

```
>>> #simple plot with legend
... import numpy as np  #It will be useful to deal with data
>>> import matplotlib.pyplot as pl  #procedural interface
>>>
>>> X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
>>> C, S = np.cos(X), np.sin(X)
>>> pl.plot(X, C, label=r'$\cos(\theta)$')
[<matplotlib.lines.Line2D object at 0x40e2690>]
>>> pl.plot(X, S, label=r'$\sin(\theta)$')
[<matplotlib.lines.Line2D object at 0x3442550>]
>>> pl.legend(loc='upper left')
<matplotlib.legend.Legend object at 0x40e28d0>
>>>
>>> pl.show()
```

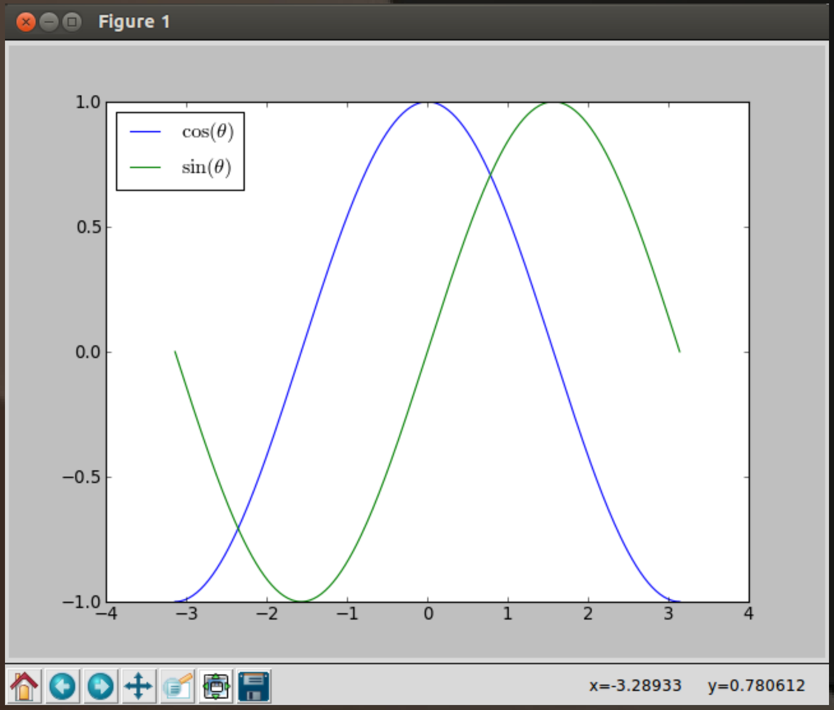
Simply using label in the plot.

You can use latex with the sintaxe:
r' $latex_keyword$'

Then you simple call legend()

You can use the variable loc to set the location of the legend in the plot

```
The location codes are

  ===============   =============
  Location String   Location Code
  ===============   =============
  'best'            0
  'upper right'     1
  'upper left'      2
  'lower left'      3
  'lower right'     4
  'right'           5
  'center left'     6
  'center right'    7
  'lower center'    8
  'upper center'    9
  'center'          10
  ===============   =============
```



Using Matplotlib

Objects in Matplotlib

The awareness of the objects will allow a proper control of the plots that you want to create.

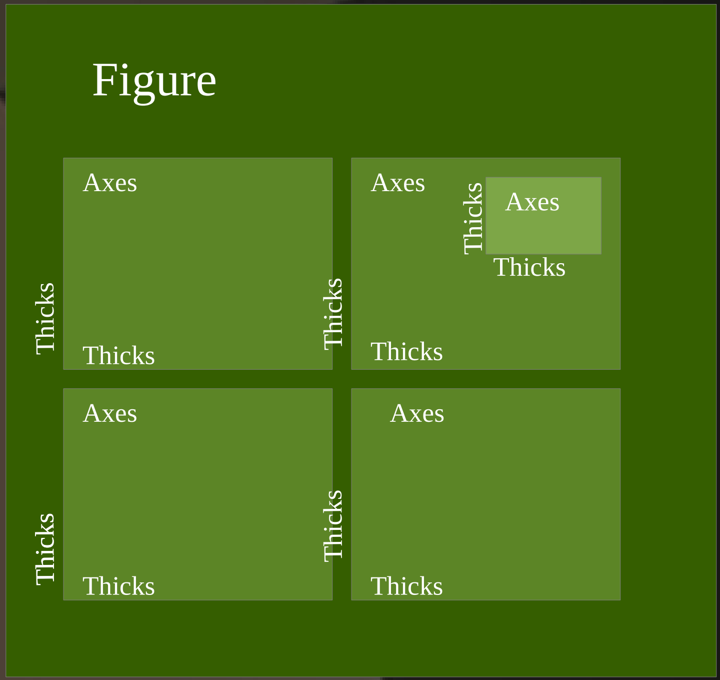
The main objects that you should know:

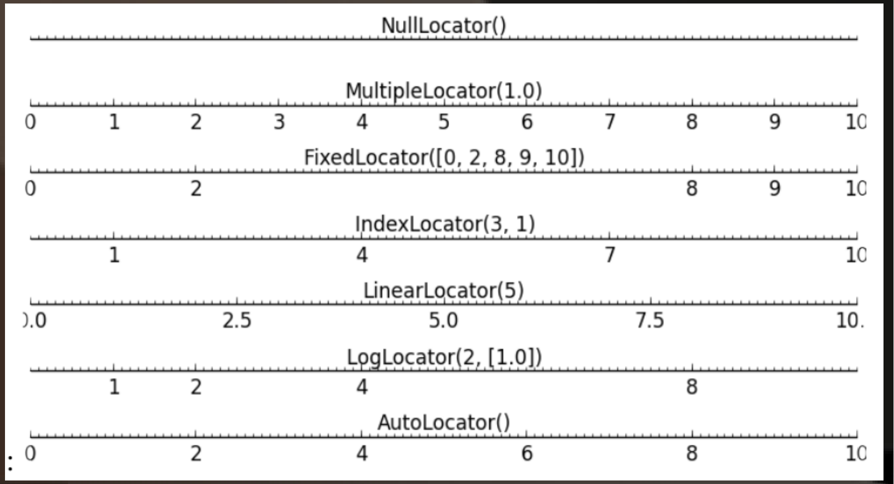
→**Figure:** A Figure in matplotlib corresponds to the whole window in the user interface.

→**Axes:** The axes is the object where you will draw your plot. You can freely control the position of the axes. Within the figure you can create/add axes.

→**Subplot:** This are actually a specific type of axes, where the positions are fixed on the figure. Within the figure you create subplots. You can create a single subplot, or a grid of subplots within the figure.

→**Thicks:** Are the objects that control each axe coordinate, uncluding the type of numbers, scale, intervals, etc...





Using Matplotlib

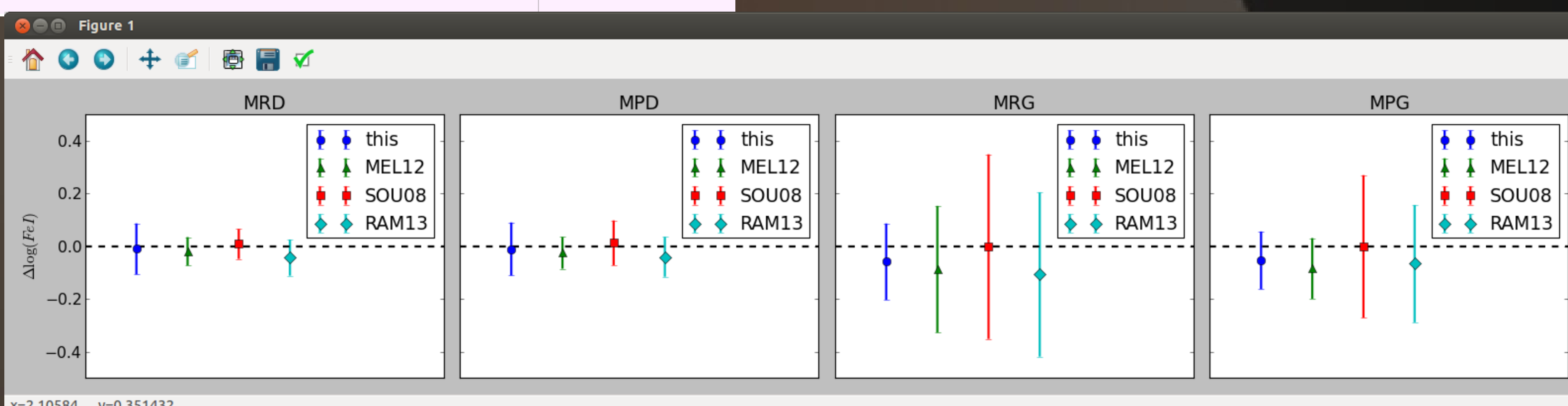
Objects in Matplotlib – An example

```python
def plot_ax_star(star,result,axe):
  linelists = [('_gesEW_mar',1,'this','o'),('_melendez_kur',2,'MEL12','^'),
              \('_mine_kur' ,3,'SOU08','s'),('_ramirez_mar' ,4,'RAM13','D')]
  axe.plot([0,10],[0,0], linestyle='--', color='k', linewidth=2.0)
  for linelist,i,labelstr,mark in linelists:
    idstr = star+linelist+'.moog'
    res = [ (file_r,star_r,nfei_r, dfei_r, sfei_r) \
            for (file_r,star_r,nfei_r, dfei_r, sfei_r) \
            in result if file_r == idstr]
    (file_r,star_r,nfei_r, dfei_r, sfei_r) = res[0]
    axe.errorbar([i], [dfei_r], yerr=sfei_r,fmt=mark, label=labelstr, linewidth=2.0, markersize=8)
  axe.set_xlim(0,7)
  axe.set_ylim(-0.5,0.5)
  axe.xaxis.set_visible(False)
  axe.set_title(star)
  axe.legend()


def plot_graphs(result):
  plt.rcParams.update({'font.size': 14})

  fig = plt.figure(figsize=(20, 4))
  ax1= fig.add_subplot(141)
  ax1.set_ylabel(r'$\Delta \log(FeI)$')
  ax2= fig.add_subplot(142,sharey=ax1)
  plt.setp(ax2.get_yticklabels(), visible=False)
  ax3= fig.add_subplot(143,sharey=ax1)
  plt.setp(ax3.get_yticklabels(), visible=False)
  ax4= fig.add_subplot(144,sharey=ax1)
  plt.setp(ax4.get_yticklabels(), visible=False)

  plot_ax_star('MRD',result,ax1)
  plot_ax_star('MPD',result,ax2)
  plot_ax_star('MRG',result,ax3)
  plot_ax_star('MPG',result,ax4)

#  fig.subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)
  fig.tight_layout()

  plt.show()
```

Main objects:
→ 1 Figure
→ 4 Axes constructed with subplots
→ 8 Thicks automatically constructed

Extra stuff in this example:

→ Shared axes
→ Changing font size
→ Making axes invisible
→ Using latex in lables
→ Using different symbols and automatic colors
→ Automatic legends
→ Using error bars



Using Matplotlib

Objects in Matplotlib – An example

```
def plot_graphs(result):
    plt.rcParams.update({'font.size': 14})

    fig = plt.figure(figsize=(20, 4))
    ax1= fig.add_subplot(141)
    ax1.set_ylabel(r'$\Delta \log(FeI)$')
    ax2= fig.add_subplot(142,sharey=ax1)
    plt.setp(ax2.get_yticklabels(), visible=False)
    ax3= fig.add_subplot(143,sharey=ax1)
    plt.setp(ax3.get_yticklabels(), visible=False)
    ax4= fig.add_subplot(144,sharey=ax1)
    plt.setp(ax4.get_yticklabels(), visible=False)

    plot_ax_star('MRD',result,ax1)
    plot_ax_star('MPD',result,ax2)
    plot_ax_star('MRG',result,ax3)
    plot_ax_star('MPG',result,ax4)

#   fig.subplots_adjust(left=None, bottom=None, right=
    fig.tight_layout()

    plt.show()
```

Main objects:
→ 1 Figure
→ 4 Axes constructed with subplots

Extra stuff in this example:

→ Changing font size
→ Using latex in lables
→ Shared axes
→ Making axes invisible

Calling a specific function to create each individual plot

Using Matplotlib

Objects in Matplotlib

Only difference is the different data to be plotted

This function does the same plot for each axes.

```
def plot_ax_star(star,result,axe):
  linelists = [('_gesEW_mar',1,'this','o'),('_melendez_kur',2,'MEL12','^'),\
               ('_mine_kur' ,3,'SOU08','s'),('_ramirez_mar' ,4,'RAM13','D')]
  axe.plot([0,10],[0,0], linestyle='--', color='k', linewidth=2.0)
  for linelist,i,labelstr,mark in linelists:
    idstr = star+linelist+'.moog'
    res = [ (file_r,star_r,nfei_r, dfei_r, sfei_r) \
            for (file_r,star_r,nfei_r, dfei_r, sfei_r) \
            in result if file_r == idstr]
    (file_r,star_r,nfei_r, dfei_r, sfei_r) = res[0]
    axe.errorbar([i], [dfei_r], yerr=sfei_r,fmt=mark, label=labelstr, \
                 linewidth=2.0, markersize=8)
  axe.set_xlim(0,7)
  axe.set_ylim(-0.5,0.5)
  axe.xaxis.set_visible(False)
  axe.set_title(star)
  axe.legend()
```

Extra stuff in this example:

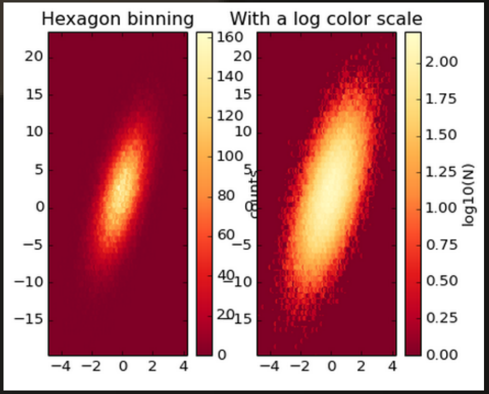
→ Horizontal dashed line
→ Using different symbols and automatic colors
→ Automatic legends
→ Using error bars

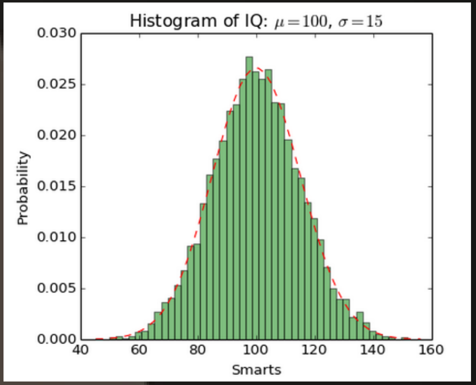
Matplotlib

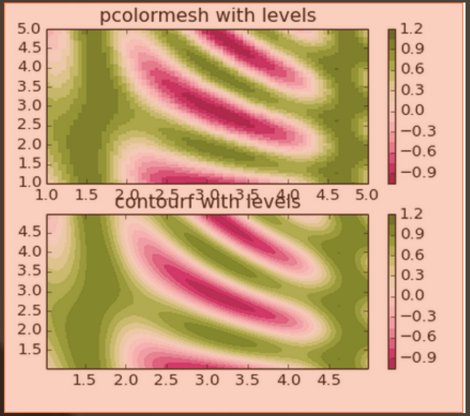
Matplotlib is a huge module which contains many features and many possibilities.

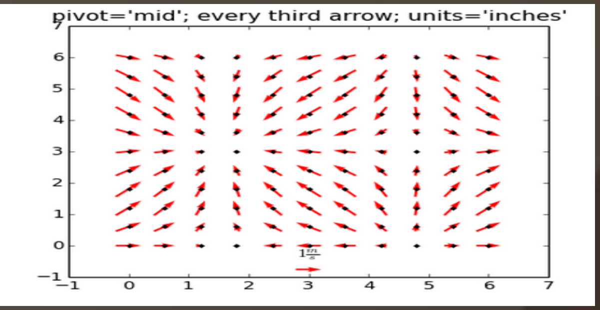
The use of Objects is benefit to control your plot easily. Experience comes with practice

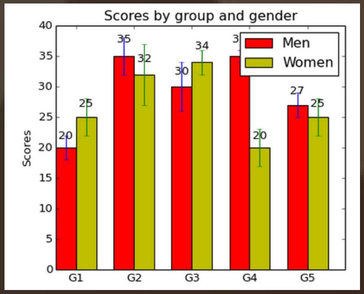
The is a gallery with several examples and source code in Matplotlib: http://matplotlib.org/gallery.html (You can use it for your inspiration...)

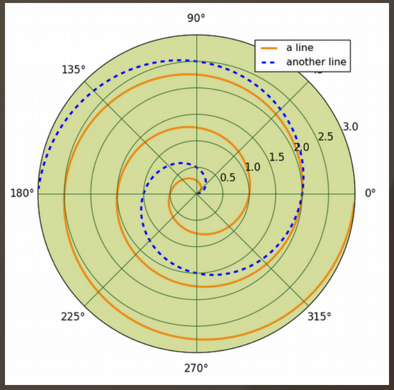


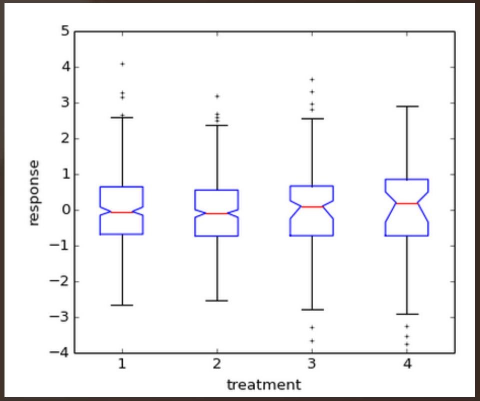


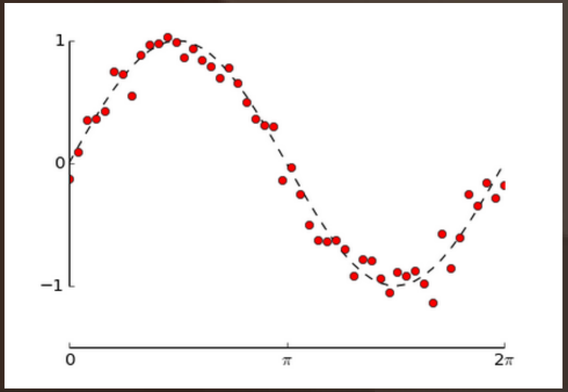


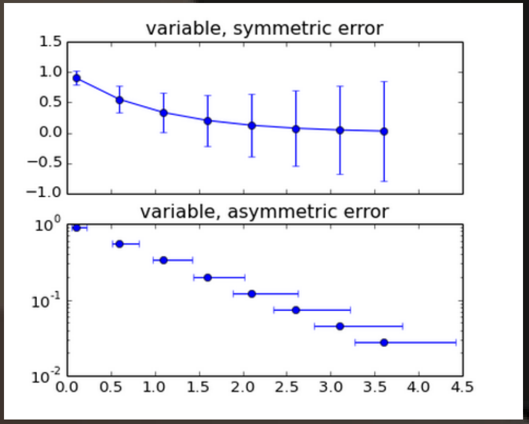




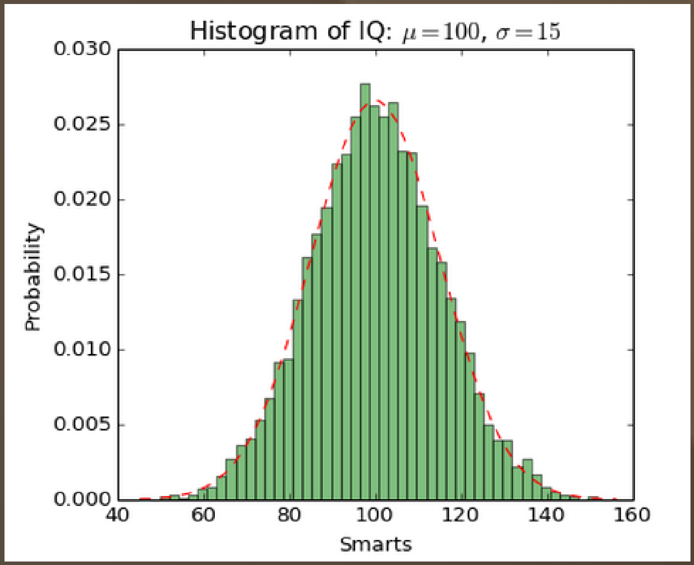








Matplotlib – Gallery Examples




```
"""
Demo of the histogram (hist) function with a few features.

In addition to the basic histogram, this demo shows a few optional features:

    * Setting the number of data bins
    * The ``normed`` flag, which normalizes bin heights so that the integral of
      the histogram is 1. The resulting histogram is a probability density.
    * Setting the face color of the bars
    * Setting the opacity (alpha value).

"""
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt


# example data
mu = 100 # mean of distribution
sigma = 15 # standard deviation of distribution
x = mu + sigma * np.random.randn(10000)

num_bins = 50
# the histogram of the data
n, bins, patches = plt.hist(x, num_bins, normed=1, facecolor='green', alpha=0.5)
# add a 'best fit' line
y = mlab.normpdf(bins, mu, sigma)
plt.plot(bins, y, 'r--')
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title(r'Histogram of IQ: $\mu=100$, $\sigma=15$')

# Tweak spacing to prevent clipping of ylabel
plt.subplots_adjust(left=0.15)
plt.show()
```

Random Gaussian Data Generation

Defining number of bins

Plotting histogram

Getting a normal probability density function. help(mlab)
(Mlab is a module for Numerical python functions written for compatability with MATLAB commands with the same names.)

Plotting pdf

Setting titles

Adjusting subplot in Figure

Matplotlib – Gallery Examples

```
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
import numpy as np
from matplotlib.mlab import bivariate_normal


N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]

# A low hump with a spike coming out of the top right.
# Needs to have z/colour axis on a log scale so we see both hump and spike.
# linear scale only shows the spike.
Z1 = bivariate_normal(X, Y, 0.1, 0.2, 1.0, 1.0) + 0.1 * bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)

plt.subplot(2,1,1)
plt.pcolor(X, Y, Z1, norm=LogNorm(vmin=Z1.min(), vmax=Z1.max()), cmap='PuBu_r')
plt.colorbar()

plt.subplot(2,1,2)
plt.pcolor(X, Y, Z1, cmap='PuBu_r')
plt.colorbar()


plt.show()
```

Normalize a given value to the 0-1 range on a log scale

Create the parameter space grid for the surface plot

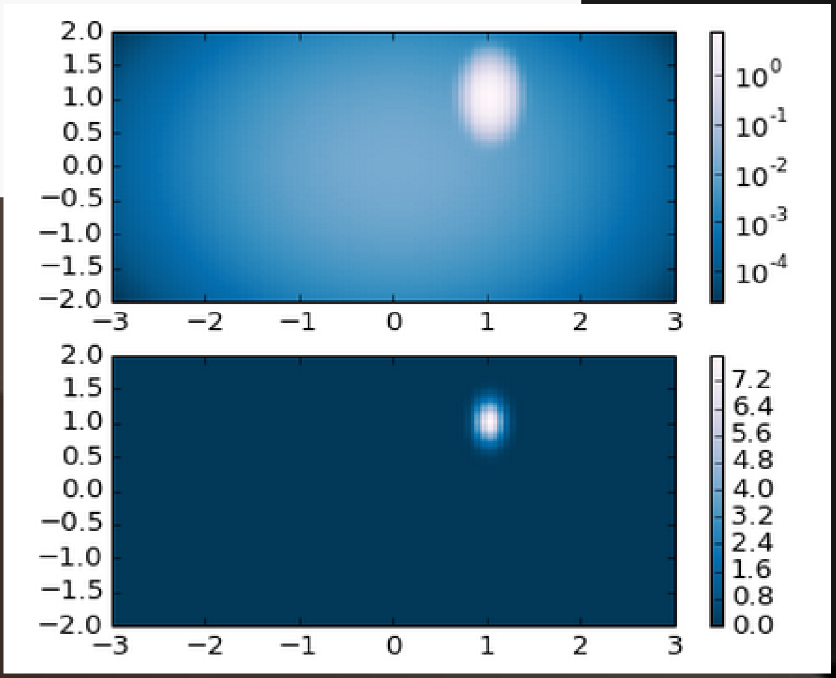
Creating data from bivariate normal distributions

This function does the same plot for each axes.

Creating the subplots – a grid of 2 rows, 1 column

Ploting a colar map

Creating a colar bar



Matplotlib – Gallery Examples

Necessary Imports

Changing the thick direction at the x and y axes.

Generating the random data

Creating the figure

Ploting the contours

Defining locations manually

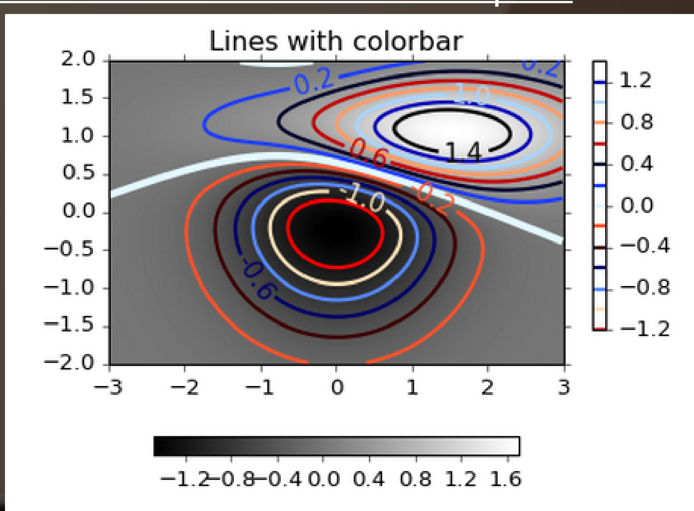
Plotting labels on contours

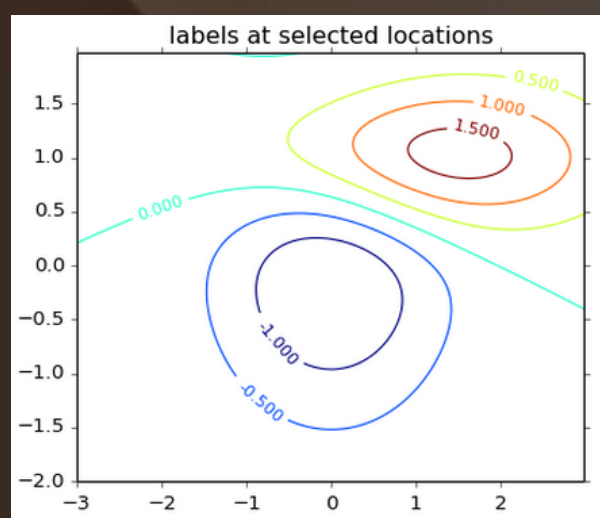
Adding title

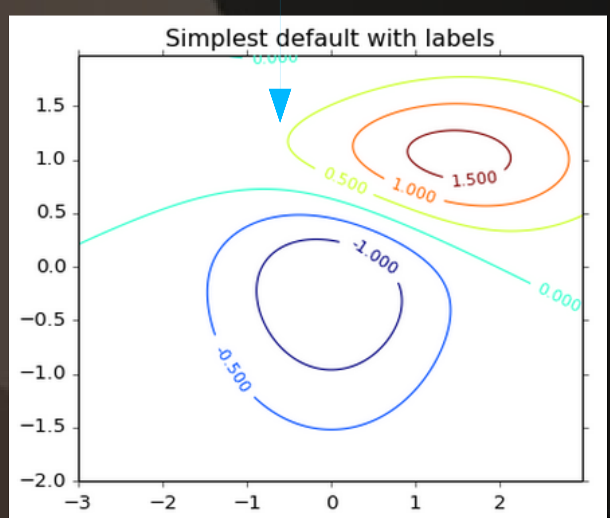
```
import matplotlib
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

matplotlib.rcParams['xtick.direction'] = 'out'
matplotlib.rcParams['ytick.direction'] = 'out'

delta = 0.025
x = np.arange(-3.0, 3.0, delta)
y = np.arange(-2.0, 2.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
# difference of Gaussians
Z = 10.0 * (Z2 - Z1)
# contour labels can be placed manually by providing list of positions
# (in data coordinate). See ginput_manual_clabel.py for interactive
# placement.
plt.figure()
CS = plt.contour(X, Y, Z)
manual_locations = [(-1, -1.4), (-0.62, -0.7), (-2, 0.5), (1.7, 1.2), (2.0, 1.4), (2.4, 1.7)]
plt.clabel(CS, inline=1, fontsize=10, manual=manual_locations)
plt.title('labels at selected locations')
```

Ignoring manual locations you get this plot

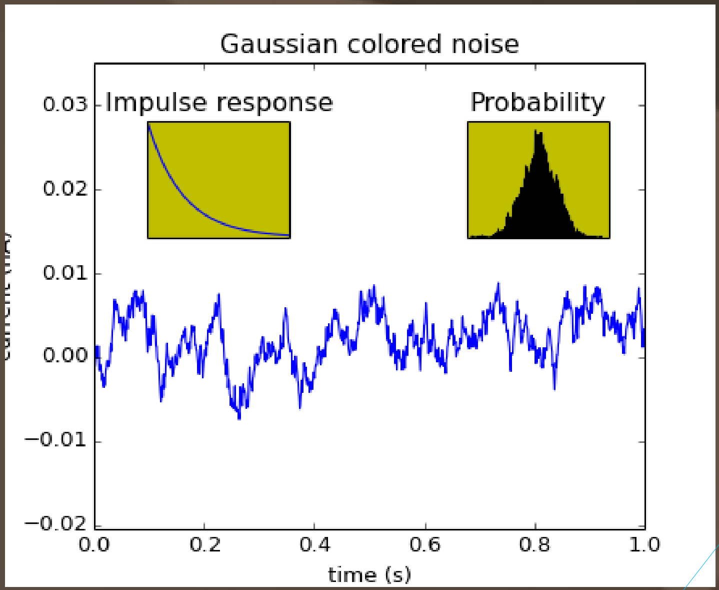
You have more advanced examples:







Matplotlib – Gallery Examples



Example using pylab directly:

Generating the random data

Ploting main axes.
(Figure is automatically created)

Changing limits for x & y axes(thicks)

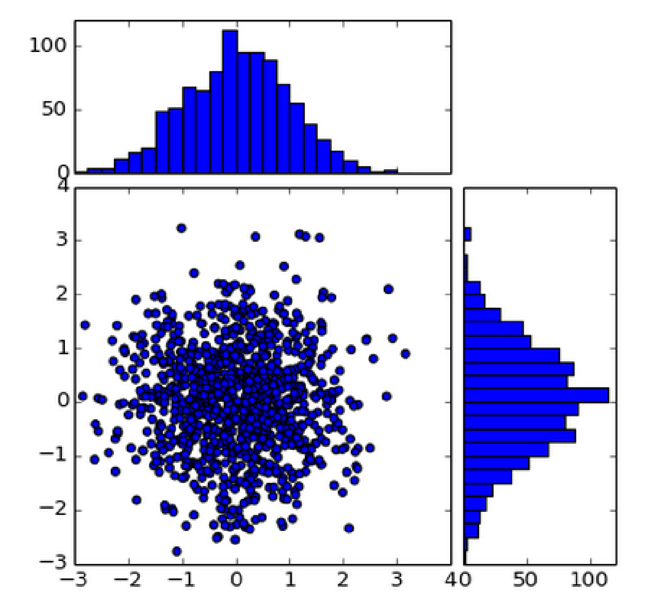
Setting titles

Creating 1st axes inside – top right
(yellow background)
Plot Histogram
Empty thicks from axes with setp()

Creating 2nd axes inside – top left
Simple plot
Empty thicks from axes and setting x limit

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

from pylab import *

# create some data to use for the plot
dt = 0.001
t = arange(0.0, 10.0, dt)
r = exp(-t[:1000]/0.05)               # impulse response
x = randn(len(t))
s = convolve(x,r)[:len(x)]*dt  # colored noise

# the main axes is subplot(111) by default
plot(t, s)
axis([0, 1, 1.1*amin(s), 2*amax(s) ])
xlabel('time (s)')
ylabel('current (nA)')
title('Gaussian colored noise')

# this is an inset axes over the main axes
a = axes([.65, .6, .2, .2], axisbg='y')
n, bins, patches = hist(s, 400, normed=1)
title('Probability')
setp(a, xticks=[], yticks=[])

# this is another inset axes over the main axes
a = axes([0.2, 0.6, .2, .2], axisbg='y')
plot(t[:len(r)], r)
title('Impulse response')
setp(a, xlim=(0,.2), xticks=[], yticks=[])


show()
```

Matplotlib – Gallery Examples



Import modules
Need mpl_toolkits.axes
Random data creation
Figure and axes created with subplot
Create a scater plot
Set equal scale for xy
Creation of the 2 attached axes for xy
Making x axes invisible in attached axes
Defining limits for the histograms
Defining bins
Plotting histograms in each attached axes
Adjusting thicks manually
Making invisible the default ones
And add thicks manually

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

# the random data
x = np.random.randn(1000)
y = np.random.randn(1000)


fig, axScatter = plt.subplots(figsize=(5.5,5.5))

# the scatter plot:
axScatter.scatter(x, y)
axScatter.set_aspect(1.)

# create new axes on the right and on the top of the current axes
# The first argument of the new_vertical(new_horizontal) method is
# the height (width) of the axes to be created in inches.
divider = make_axes_locatable(axScatter)
axHistx = divider.append_axes("top", 1.2, pad=0.1, sharex=axScatter)
axHisty = divider.append_axes("right", 1.2, pad=0.1, sharey=axScatter)

# make some labels invisible
plt.setp(axHistx.get_xticklabels() + axHisty.get_yticklabels(),
         visible=False)

# now determine nice limits by hand:
binwidth = 0.25
xymax = np.max( [np.max(np.fabs(x)), np.max(np.fabs(y))] )
lim = ( int(xymax/binwidth) + 1) * binwidth

bins = np.arange(-lim, lim + binwidth, binwidth)
axHistx.hist(x, bins=bins)
axHisty.hist(y, bins=bins, orientation='horizontal')

# the xaxis of axHistx and yaxis of axHisty are shared with axScatter,
# thus there is no need to manually adjust the xlim and ylim of these
# axis.

#axHistx.axis["bottom"].major_ticklabels.set_visible(False)
for tl in axHistx.get_xticklabels():
    tl.set_visible(False)
axHistx.set_yticks([0, 50, 100])

#axHisty.axis["left"].major_ticklabels.set_visible(False)
for tl in axHisty.get_yticklabels():
    tl.set_visible(False)
axHisty.set_xticks([0, 50, 100])

plt.draw()
plt.show()
```

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)

Why Python?