UC Berkeley Extension

Python for Data Analysis and Scientific Computing

X433.3 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

Course Content Outline

- **Introduction to Python®**
- Python - pros and cons
- Installing the environment with core packages
- Python modules, packages and scientific blocks
- Working with the shell, IPyton and the editor HW1

- **Basic language specifics 1/2**
- Basic arithmetic operations, assignment operators, data types, containers
- Iterative programming (if/elif/else)
- Conditional expressions
- Recursion programming (for/continue/while/break)
- Functions: definition, return values, local vs. global variables

- **Basic language specifics 2/2**
- Classes / Functions (cont.): objects, methods, passing by value and reference
- Scripts, modules, packages
- I/O interaction with files
- Standard library
- Exceptions

- **NumPy 1/3**
- Why NumPy?
- Data type objects
- NumPy arrays
- Indexing and slicing of arrays HW2

- **Matplotlib**
- What is Matplotlib?
- Basic plotting
- Tools: title, labels, legend, axis, points, subplots, etc.
- Advanced plotting: scatter, pie, bar, 3D plots, etc. HW3

Lecture 4 exercise discussion

- Class exercise

Solution:

```python
from numpy import array as ar
from numpy import matrix as mx
from numpy import int16, random

class _file_operations():
    def write_my_file(A):
        file = open('my_array','w')
        file.writelines('%s' %str(A))
        file.close()

    def read_my_file():
        file = open('my_array','r')
        B = file.read()
        file.close()
        return B

B = random.randint(5,45,6)
B.tofile('my_array', sep=',', format='%s')
B = _file_operations.read_my_file()
B = ar(B.split(','), dtype=int16)
C = mx([[3],[2],[5]])
D = C*B
_file_operations.write_my_file(D)
print(_file_operations.read_my_file())
```

Lecture 4 exercise discussion

- Class exercise

Solution:

```
In [1]: B = random.randint(5,45,6)

In [2]: B
Out[2]: array([21, 43, 16, 26, 43, 18])

In [3]: B.tofile('my_array', sep=',', format='%s')

In [4]: B = _file_operations.read_my_file()

In [5]: B
Out[5]: '21,43,16,26,43,18'

In [6]: type(B)
Out[6]: str

In [7]: B = ar(B.split(','), dtype=int16)

In [8]: B
Out[8]: array([21, 43, 16, 26, 43, 18], dtype=int16)

In [9]: C = mx([[3],[2],[5]])

In [10]: C
Out[10]:
matrix([[3],
        [2],
        [5]])

In [11]: D = C*B

In [12]: D
Out[12]:
matrix([[ 63, 129,  48,  78, 129,  54],
        [ 42,  86,  32,  52,  86,  36],
        [105, 215,  80, 130, 215,  90]])
```

HW2 – discussion

- HW2 – discussion

```
# 1. Include a section line with your name:
## HW 2: Alexander Iliev
from numpy import matrix, array, random, min, max

# 2. Create Matrix A with size (3,5) containing random numbers:
A = random.random(15)
A = A.reshape(3,5)
A = matrix(A)

# 3. Find the size and length of Matrix A:
A.size
len(A)

# 4. Resize (crop) Matrix A to size (3,4):
A = A[0:3,0:4]

# 5. Find the transpose of Matrix A and record it as Matrix B:
B = A.T

# 6. Find the minimum value in column 1 of each row of Matrix B:
B[:,1].min()

# 7. Find the minimum and maximum values for the entire Matrix A:
A.min()
A.max()

# 8. Create Vector X (an array) with 4 random numbers:
X = array([random.random(4)])

# 9. Create a function and pass Vector X and Matrix A in it:
def function_HW2(a,b):
    return a*b.T

# 10. In the new function multiply Vector X with Matrix A
# and return the result back to the main program (result D):
D = function_HW2(X,A)

# 11. Create a complex number Z with absolute and real parts ~= 0:
Z = 6+5j

# 12. Show its real and imaginary parts:
Z.real
Z.imag
abs(Z)

# 13. Multiply result D with the absolute value of Z and record it to C:
C = D*abs(Z)

# 14. Convert Matrix B from a matrix to a string and overwrite B:
B = str(B)

# 15. Display a text on the screen: ‘I am done with HW2‘,
# but pass ‘HW2’ as a string variable:
print('%s is done with HW2' %'Alex')
```

Numpy

Recap:

- What are matrix, array and ndarray in NumPy

```
## Recap: What are matrix, array and ndarray in NumPy:
from numpy import matrix, array, ndarray, int16, dot

# Arrays should be constructed using `array`, `zeros` or `empty`:
A = array([[2,3,4],[4,5,6]])

# Construct and assign a mtraix:
B = matrix([[2,3,4],[4,5,6]])

# Construct an empty array:
C = ndarray([2,3], dtype=int16)

# Assign:
C[0,:] = A[0,:]
C[1,:] = B[1,:]

A*A
B*B.T
C*C

# To get matrix multiplication of an ndarray:
dot(A,A.T)
dot(C,C.T)
```

Numpy

Recap:

- What are matrix, array and ndarray in NumPy

```
In [1]: from numpy import matrix, array, ndarray, int16, dot

In [2]: A = array([[2,3,4],[4,5,6]])

In [3]: B = matrix([[2,3,4],[4,5,6]])

In [4]: C = ndarray([2,3], dtype=int16)

In [5]: A
Out[5]:
array([[2, 3, 4],
       [4, 5, 6]])

In [6]: B
Out[6]:
matrix([[2, 3, 4],
        [4, 5, 6]])

In [7]: C
Out[7]:
array([[    0,     0,     0],
       [    0, 15653,  4166]], dtype=int16)

In [8]: C[0,:] = A[0,:]

In [9]: C[1,:] = B[1,:]

In [10]: C
Out[10]:
array([[2, 3, 4],
       [4, 5, 6]], dtype=int16)
```

```
In [11]: type(A)
Out[11]: numpy.ndarray

In [12]: type(C)
Out[12]: numpy.ndarray

In [13]: A*A
Out[13]:
array([[ 4,  9, 16],
       [16, 25, 36]])

In [14]: B*B.T
Out[14]:
matrix([[29, 47],
        [47, 77]])

In [15]: C*C
Out[15]:
array([[ 4,  9, 16],
       [16, 25, 36]], dtype=int16)

In [16]: dot(A,A.T)
Out[16]:
array([[29, 47],
       [47, 77]])

In [17]: dot(C,C.T)
Out[17]:
array([[29, 47],
       [47, 77]], dtype=int16)
```

What is Matplotlib?

- What is Matplotlib?
  - Matplotlib is an open source advanced plotting library designed to support interactive high quality plotting
  - Matplotlib was created by John Hunter (1968-2012) – http://matplotlib.org/
  - there are many different packages that offer advanced 2D and 3D functionality, but Matplotlib is probably the single status quo graphical package for Python
  - its syntax is similar to the one Matlab uses, which was one of the goals when Matplotlib was built
  - it provides an object oriented easy to use interface
  - the Matplotlib library can create: *simple plots, bar charts, histograms, power spectrum visualizations, error charts, scatter plots* and much more
  - Matplotlib has an interactive mode that supports multiple windowing toolkits such as: Tkinter, GTK, Qt, etc.

What is Matplotlib?

- What is Matplotlib?
  - Matplotlib also supports multiple non-interactive backend systems like: postscript, PDF, SVG, antigrain geometry and Cairo
  - Matplotlib has several dependencies, one of which is NumPy, but Scipy is not
  - Matplotlib plots can be:
    - used in publishing material
    - embedded in GUI applications
    - used for non-interactive uses without any display in batch mode
  - There are many different ways that this package can be used in, such as:
    - in the Python and iPython shell (Pizo as well)
    - in Python scripts
    - in web application servers
    - in six graphical user interface toolkits
  - IPython and Pizo have a *pylab* mode that is designed for interactive plotting with Matplotlib

What is Matplotlib?

- What is Matplotlib?
  - In the enhanced interactive iPython (and Pizo) shell there are many interesting features, some of which include:
    - access to shell commands
    - named inputs and outputs
    - improved debugging
  - the command line argument *pylab* may be imported to begin an interactive Matplotlib session
  - *pylab* brings some of the plotting functionality in Matplotlib and provides a procedural interface to the Matplotlib object-oriented plotting library
  - *pylab* provides a Matlab-like environment for scientific computing, so most plotting commands in *pylab* have Matlab analogs and take and return similar arguments
  - after being imported, *pylab* loads most of NumPy into the namespace as well so that can mimic a Matlab environment more closely

What is Matplotlib?

- What is Matplotlib?
  - importing only *matplotlib.pyplot* is cleaner, so depending on what the user needs the two scenarios can be commonly seen:

```
In [1]: import numpy as np

In [2]: import matplotlib.pyplot as plt
```

or

```
In [3]: import pylab as plb
```

  - *pylab* brings the *pyplot* function of Matplotlib as well as most of NumPy. Using this call, one can do the following:

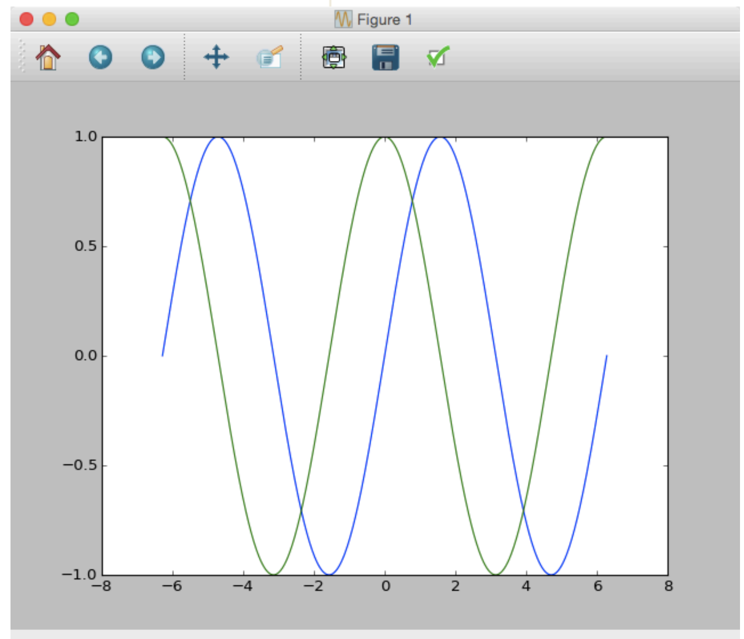
```
In [4]: a = plb.np.array([[12,34],[41,54]])

In [5]: a
Out[5]:
array([[12, 34],
       [41, 54]])

In [6]: type(a)
Out[6]: numpy.ndarray
```

so that creating an array via *pylab* is the same as creating it as if you imported NumPy

Basic plotting

- Basic plotting: *comparison*

```python
## Basic plotting using numpy and matplotlib.pyplot:
import numpy as np
import matplotlib.pyplot as plt

# lets create the array 'a' with 512 points in the range [-2*pi:2*pi]:
a = np.linspace(-np.pi*2, np.pi*2, 512, endpoint=True)

# the 'sin' and 'cos' functions have the same number of points (512):
b_sin, c_cos = np.sin(a), np.cos(a)

# lets plot the results of the two functions above:
plt.plot(a, b_sin)
plt.plot(a, c_cos)
plt.show()
```

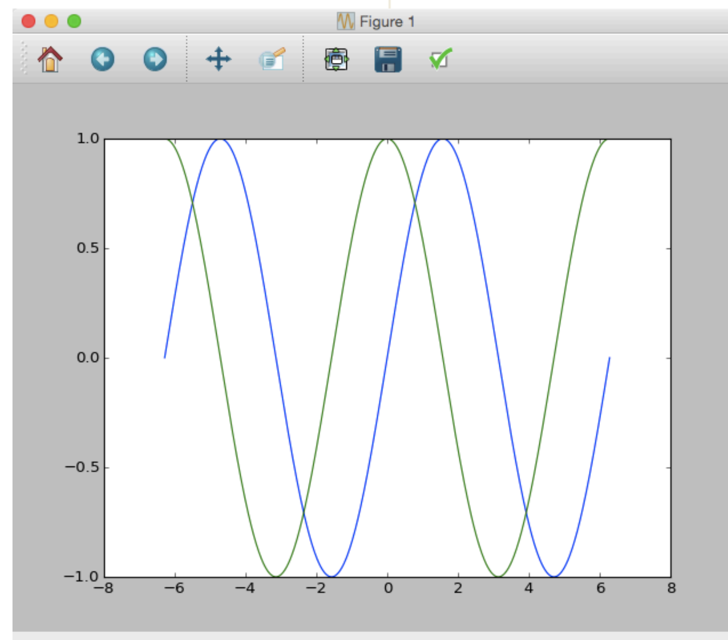


Basic plotting

- Basic plotting: *comparison*

```
## Basic plotting using pylab:
import pylab as plb

# lets create the array 'a' with 512 points in the range [-2*pi:2*pi]:
a = plb.linspace(-plb.pi*2, plb.pi*2, 512, endpoint=True)

# the 'sin' and 'cos' functions have the same number of points (512):
b_sin, c_cos = plb.sin(a), plb.cos(a)

# lets plot the results of the two functions above:
plb.plot(a, b_sin)
plb.plot(a, c_cos)
plb.show()
```

both cases produce the same result ----->



Basic plotting tools

- Basic plotting tools: *using figure size, dpi and subplots*

  – we can create a figure with a specific size and dpi (dots per inch):

  plb.figure(figsize=(10, 6), dpi=120) # this line will create an empty window

  – when two graphics are needed in the same graphic window we can use ‘subplot’:

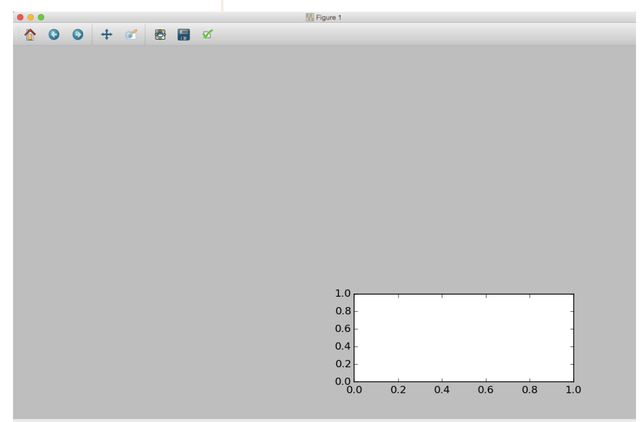
  plb.subplot(y,x,n) # will create an empty plotting space inside the window

  # where ‘y’ is the y-axis, ‘x’ is x-axis and ‘n’ is number of the

  # window to be created after setting ‘x’ and ‘y’

Example:

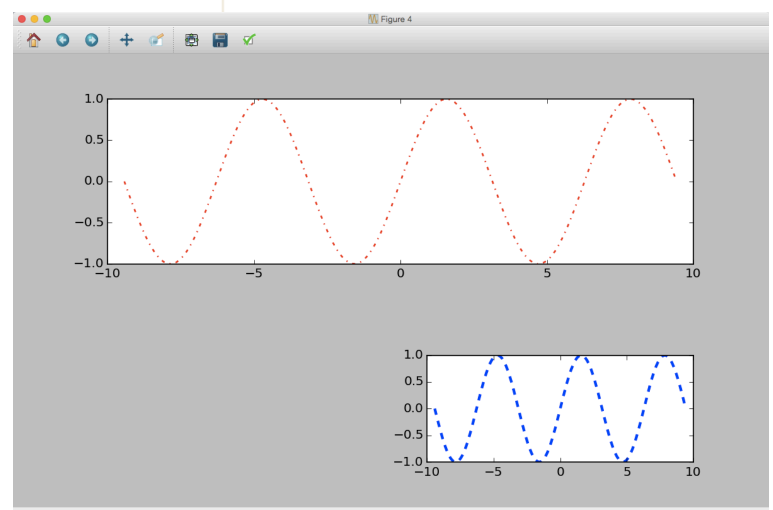
```
Python
In [7]: import pylab as plb

In [8]: plb.figure(figsize=(10, 6), dpi=120)
Out[8]: <matplotlib.figure.Figure at 0x1105f2630>

In [9]: plb.subplot(3,2,6)
Out[9]: <matplotlib.axes._subplots.AxesSubplot at 0x111196550>
```



Basic plotting tools

- Basic plotting tools: *using color, linewidth and linestyle*

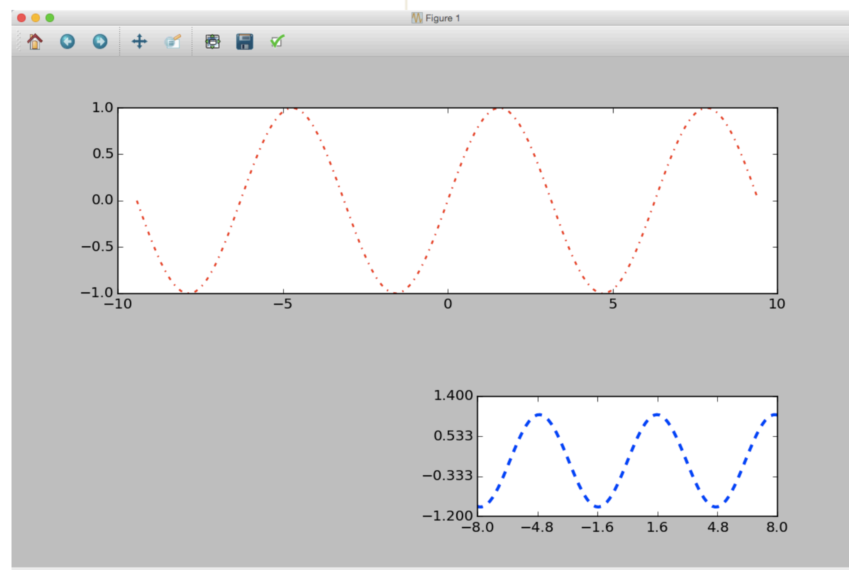
```python
## Basic plotting tools:
import pylab as plb
plb.figure(figsize=(10, 6), dpi=120)

# we create an array 'd' with 128 points in the range [-3*pi:3*pi]:
d = plb.linspace(-plb.pi*3, plb.pi*3, 128, endpoint=True)

# now we create the 'sin' and 'cos' functions from 'd' with 128 points each:
d_sin = plb.sin(d)
d_cos = plb.cos(d)

# plot 'sin' using a green dash-dotted line of width 1.5px in area (2,1,1):
plb.subplot(2,1,1)
plb.plot(d, d_sin, color="red", linewidth=1.5, linestyle="-.")

# plot 'cos' using a blue dashed line of width 1.5px in area (3,2,6):
plb.subplot(3,2,6)
plb.plot(d, d_sin, color="blue", linewidth=2.5, linestyle="--")
```



Plotting tools

- Plotting tools: *setting limits, ticks; showing and saving the plot*

```python
# we need to set the 'x' limits:
plb.xlim(-8.0, 8.0)
# then plot 'x' ticks:
plb.xticks(plb.linspace(-8, 8, 6, endpoint=True))

# now we set the 'y' limits:
plb.ylim(-1.2, 1.4)
# we set the 'y' ticks:
plb.yticks(plb.linspace(-1.2, 1.4, 4, endpoint=True))

# show the result on screen:
plb.show()

# we can now save the figure using 64 dots per inch:
plb.savefig("lecture_5.png", dpi=64)
```

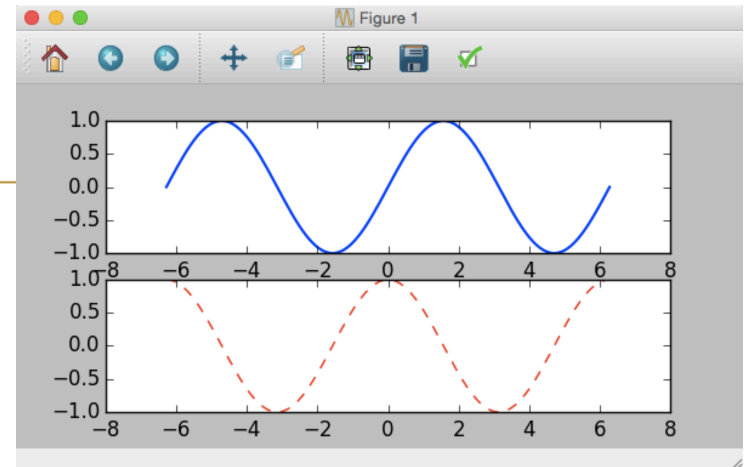


Plotting tools

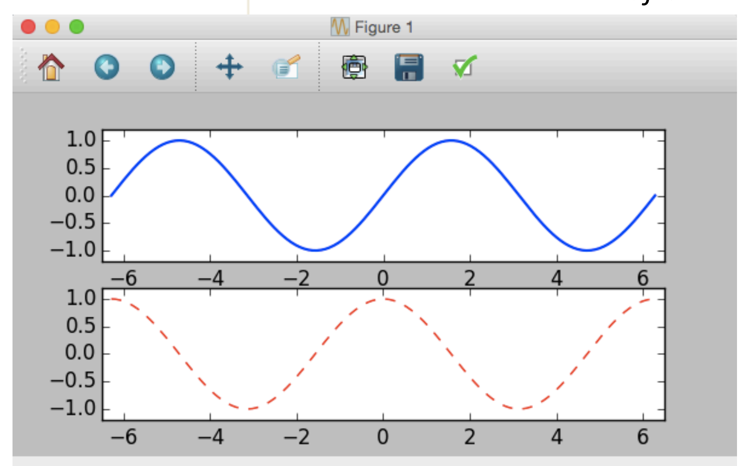
- Plotting tools: *changing plot limits*

```
## Changing plot limits:
import pylab as plb

plb.figure(figsize=(6, 3), dpi=100)
d = plb.linspace(-plb.pi*2, plb.pi*2, 128, endpoint=True)
d_sin = plb.sin(d)
d_cos = plb.cos(d)

# we now set the x,y limits for the 'sin' function:
plb.subplot(2,1,1)
plb.plot(d, d_sin, color="blue", linewidth=1.5, linestyle="-")
plb.xlim(d_sin.min() * 6.5, d_sin.max() * 6.5)
plb.ylim(d_sin.min() * 1.2, d_sin.max() * 1.2)

# below we set the x,y limits for the 'cos' function:
plb.subplot(2,1,2)
plb.plot(d, d_cos, color="red", linewidth=1, linestyle="--")
plb.xlim(d_cos.min() * 6.5, d_cos.max() * 6.5)
plb.ylim(d_cos.min() * 1.2, d_cos.max() * 1.2)
```

before



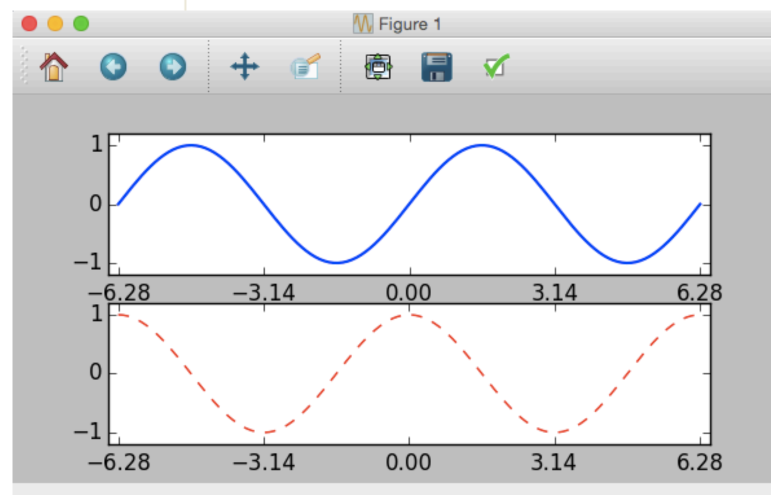
after



Plotting tools

- Plotting tools: *editing ticks*

```python
## Changing plot limits:
import pylab as plb

plb.figure(figsize=(6, 3), dpi=100)
d = plb.linspace(-plb.pi*2, plb.pi*2, 128, endpoint=True)
d_sin = plb.sin(d)
d_cos = plb.cos(d)

# we now set the x,y limits for the 'sin' function:
plb.subplot(2,1,1)
plb.plot(d, d_sin, color="blue", linewidth=1.5, linestyle="-")
plb.xlim(d_sin.min() * 6.5, d_sin.max() * 6.5)
plb.ylim(d_sin.min() * 1.2, d_sin.max() * 1.2)
plb.xticks([-plb.pi*2, -plb.pi, 0, plb.pi, plb.pi*2]) #<-----
plb.yticks([-1, 0, +1])

# below we set the x,y limits for the 'cos' function:
plb.subplot(2,1,2)
plb.plot(d, d_cos, color="red", linewidth=1, linestyle="--")
plb.xlim(d_cos.min() * 6.5, d_cos.max() * 6.5)
plb.ylim(d_cos.min() * 1.2, d_cos.max() * 1.2)
plb.xticks([-plb.pi*2, -plb.pi, 0, plb.pi, plb.pi*2]) #<-----
plb.yticks([-1, 0, +1])
```



Plotting tools

- Plotting tools: *adding tick labels*

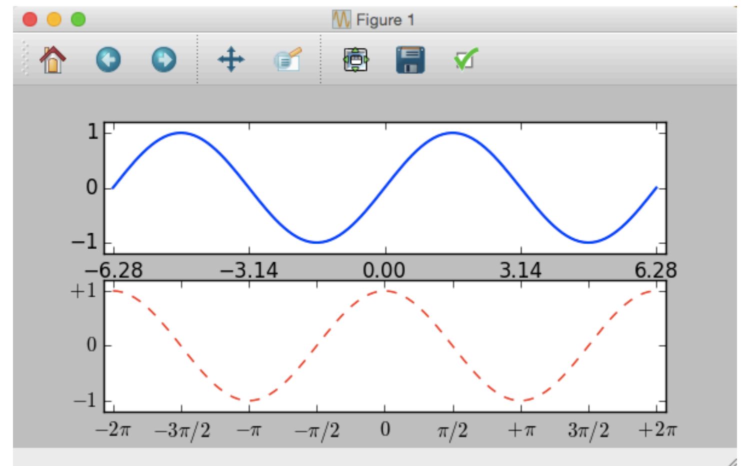
Now that we set the ticks correctly, we need to be a bit more explicit about what they represent, so we add the following code:

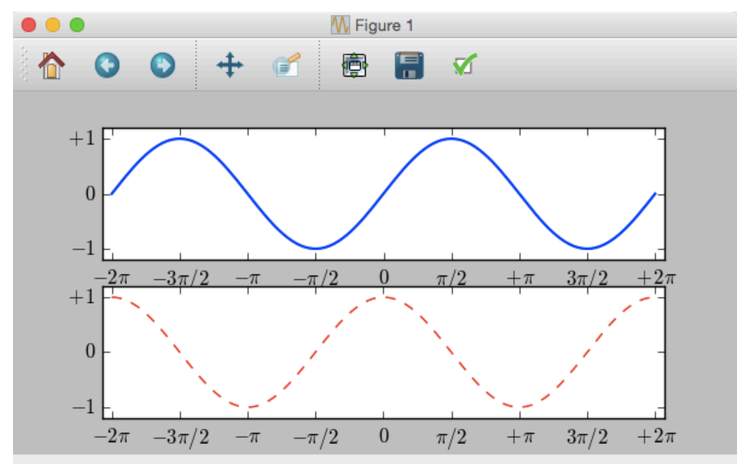
```
# adding x,y tick labels for plot (2,1,2):
plb.xticks([-plb.pi*2, -3*plb.pi/2, -plb.pi, -plb.pi/2, 0,
                 plb.pi/2, plb.pi, 3*plb.pi/2, plb.pi*2],
           ['$-2\pi$', '$-3\pi/2$', '$-\pi$', '$-\pi/2$', '$0$', \
                 '$\pi/2$', '$+\pi$', '$3\pi/2$', '$+2\pi$'])
plb.yticks([-1, 0, +1],
           ['$-1$', '$0$', '$+1$'])
```

in order to do the same for plot (2,1,1) we need to specifically request it:

```
# now adding x,y tick labels for plot (2,1,1):
plb.subplot(2,1,1)
plb.xticks([-plb.pi*2, -3*plb.pi/2, -plb.pi, -plb.pi/2, 0,
                 plb.pi/2, plb.pi, 3*plb.pi/2, plb.pi*2],
           ['$-2\pi$', '$-3\pi/2$', '$-\pi$', '$-\pi/2$', '$0$', \
                 '$\pi/2$', '$+\pi$', '$3\pi/2$', '$+2\pi$'])
plb.yticks([-1, 0, +1],
           ['$-1$', '$0$', '$+1$'])
```

line split

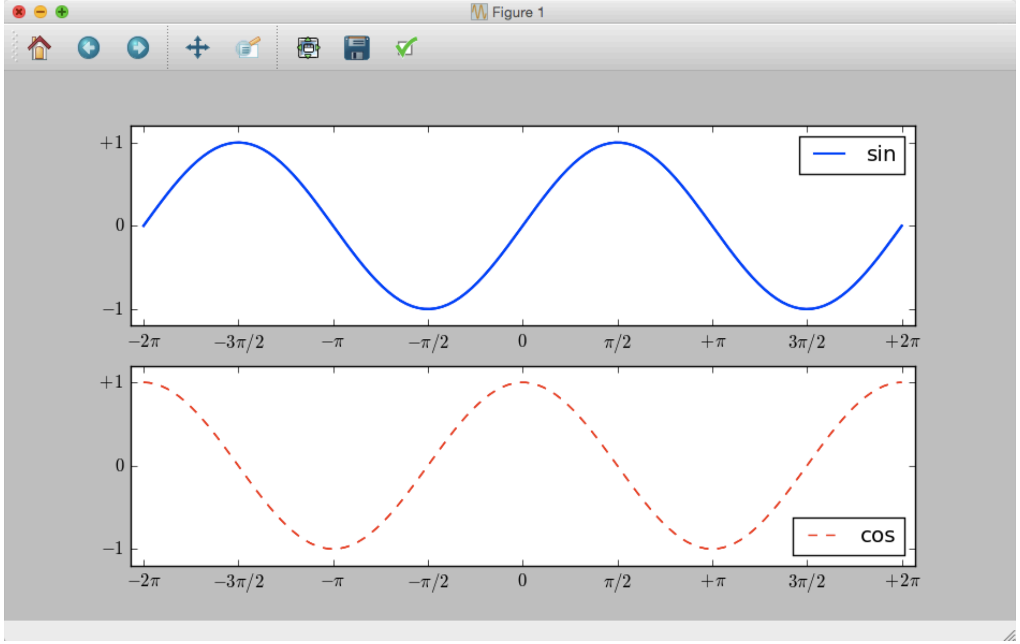




Plotting tools

- Plotting tools: *adding a legend*

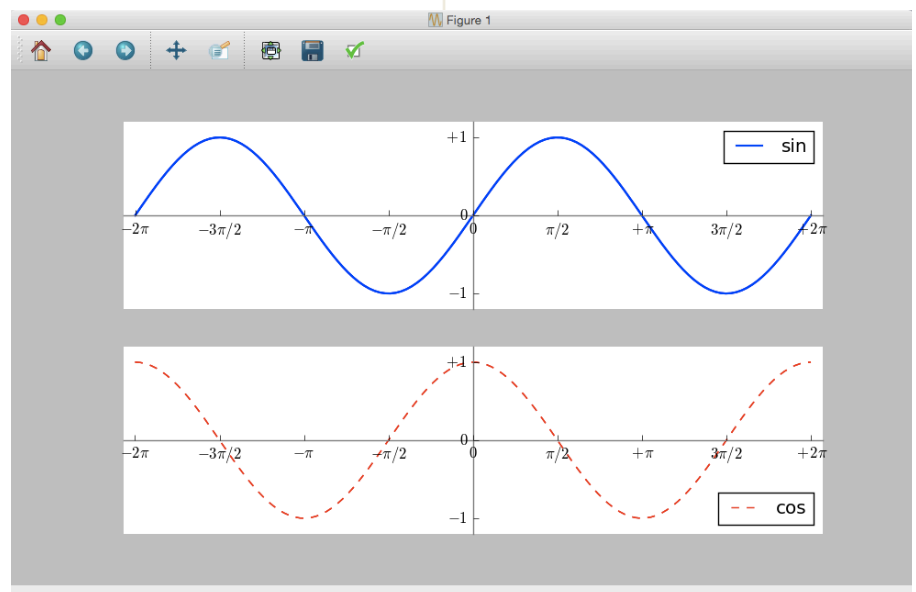
```
# adding a legend clarifying the plots:
plb.subplot(2,1,1)
plb.plot(d, d_sin, color="blue",  linewidth=1.5, linestyle="-", label="sin")
plb.legend(loc='upper right')
plb.subplot(2,1,2)
plb.plot(d, d_cos, color="red", linewidth=1, linestyle="--", label="cos")
plb.legend(loc='lower right')
```



Plotting tools

- Plotting tools: *setting the x and y axis with proper labeling*

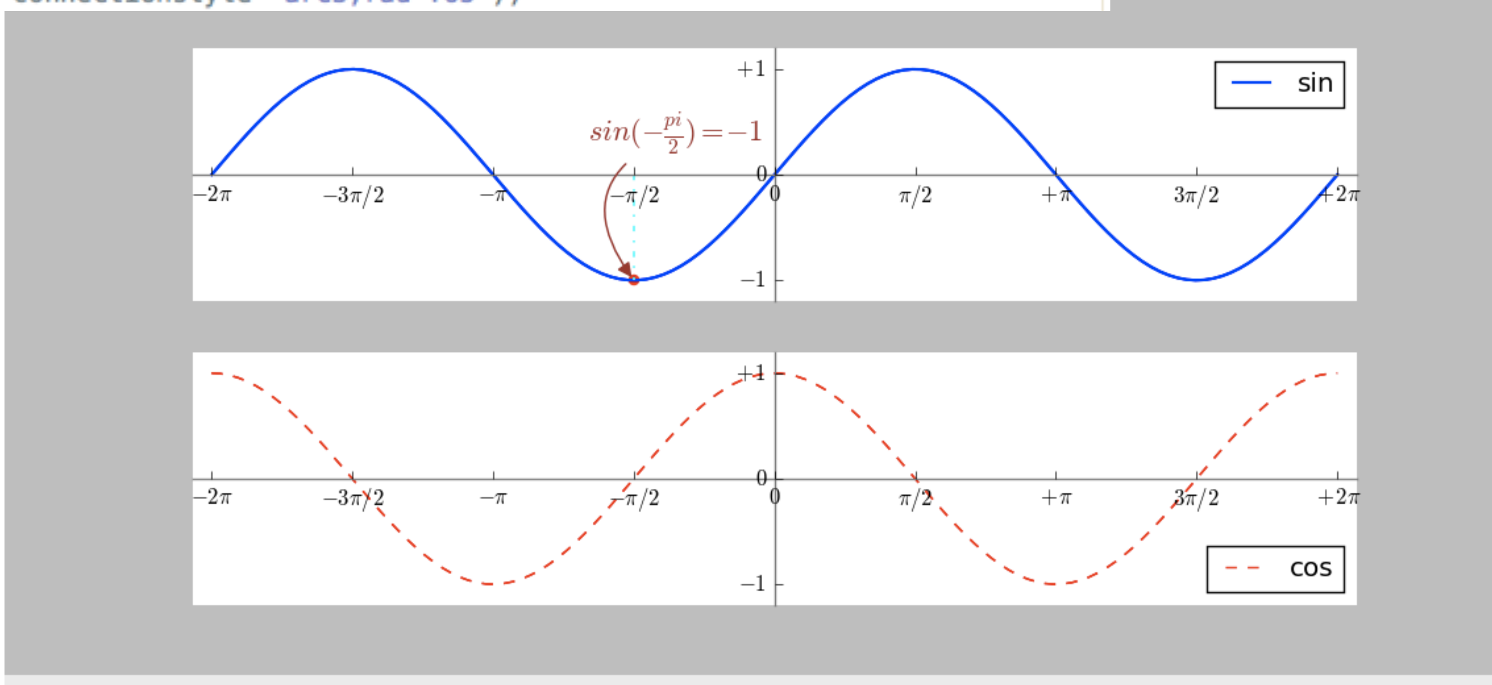
```python
# setting the axis:
ax1 = plb.gca()  # gca - 'get current axis'
ax1.spines['top'].set_color('none')    # to get rid of the black border line
ax1.spines['bottom'].set_color('none') # to get rid of the black border line
ax1.spines['left'].set_color('none')   # to get rid of the black border line
ax1.spines['right'].set_color('none')  # to get rid of the black border line
ax1.xaxis.set_ticks_position('bottom')
ax1.spines['bottom'].set_position(('data',0))
ax1.spines['bottom'].set_color('gray')
ax1.yaxis.set_ticks_position('left')
ax1.spines['left'].set_position(('data',0))
ax1.spines['left'].set_color('gray')

plb.subplot(2,1,1)
ax2 = plb.gca()  # gca - 'get current axis'
ax2.spines['top'].set_color('none')
ax2.spines['bottom'].set_color('none')
ax2.spines['left'].set_color('none')
ax2.spines['right'].set_color('none')
ax2.xaxis.set_ticks_position('bottom')
ax2.spines['bottom'].set_position(('data',0))
ax2.spines['bottom'].set_color('gray')
ax2.yaxis.set_ticks_position('left')
ax2.spines['left'].set_position(('data',0))
ax2.spines['left'].set_color('gray')
```



Plotting tools

- Plotting tools: *annotating a specific point on the plot*

```python
# annotating a specific point on the plot:
i = -plb.pi/2
plb.plot([i, i],[0, plb.sin(i)], color='cyan', linewidth=1.25, linestyle="-.")
plb.scatter([i, ],[plb.sin(i), ], 25, color='red')
plb.annotate(r'$sin(-\frac{pi}{2})=-1$',
             xy=(i, plb.sin(i)), xycoords='data', textcoords='offset points',
             xytext=(-25, +75), fontsize=16, color='brown',
             arrowprops=dict(arrowstyle="-|>", color='brown',
             connectionstyle="arc3,rad=.65"))
```

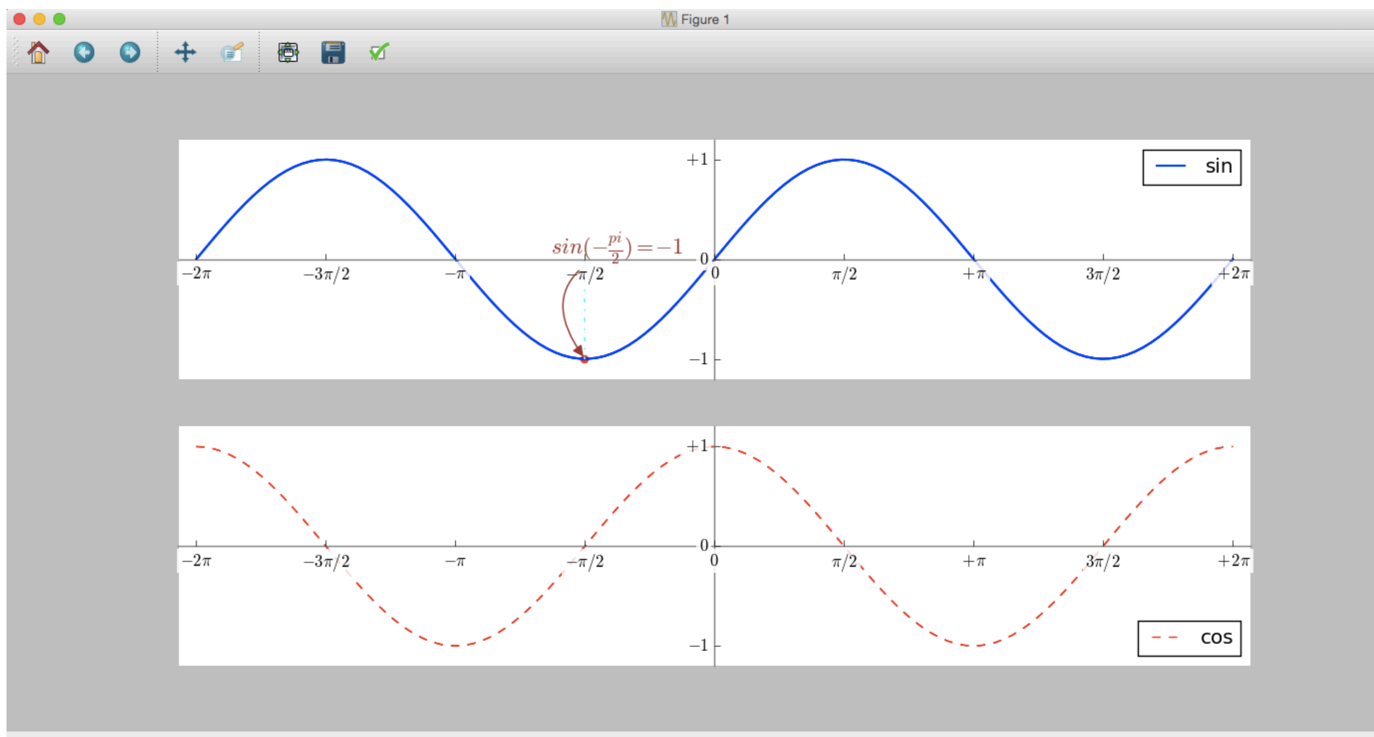


Plotting tools

- Plotting tools: *fine touches – setting label opacity (alpha)*

```python
# tick labels - fine touches:
for label in ax1.get_xticklabels() + ax1.get_yticklabels() + \
              ax2.get_xticklabels() + ax2.get_yticklabels():
    label.set_fontsize(12)
    label.set_bbox(dict(facecolor='white', edgecolor='None', alpha=0.85))
```

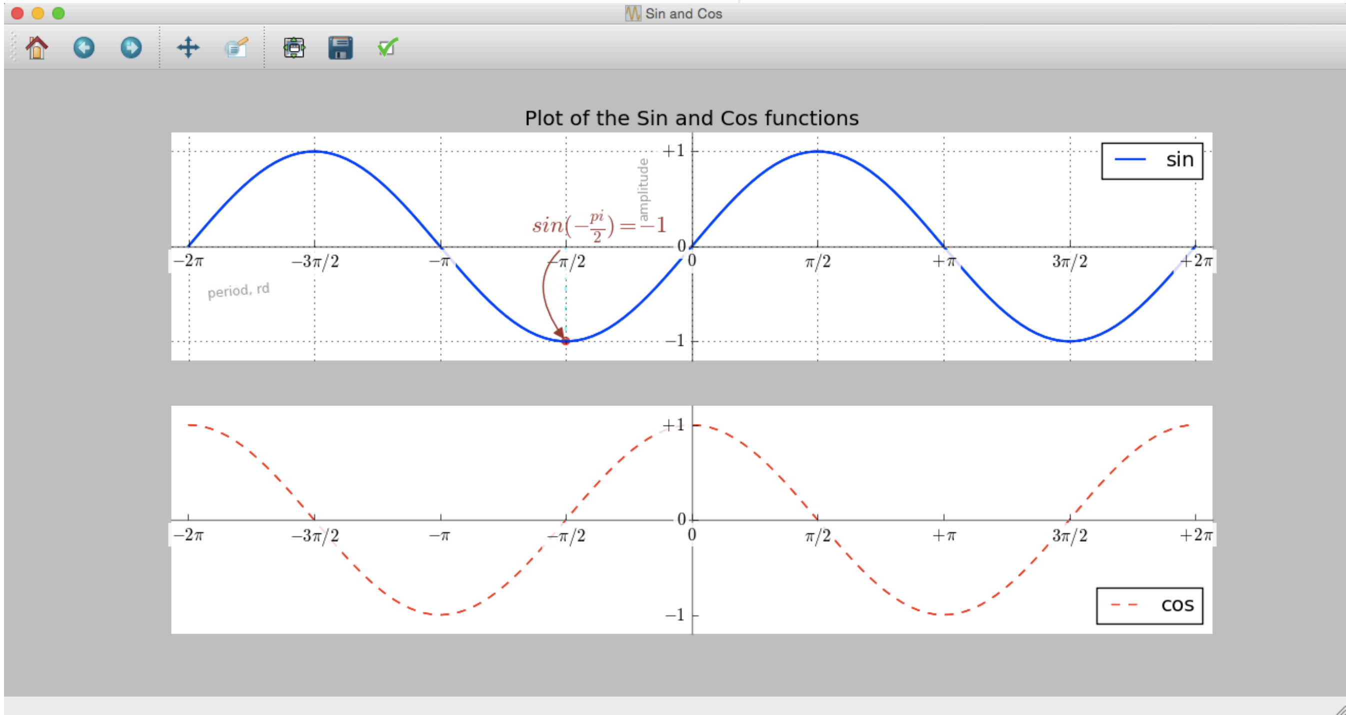
our tick labels are obscured by the plot lines running over them, so we need to make them more clear and visible



Plotting tools

- Plotting tools: *fine touches – figure name, title, x- y- labels, grid*

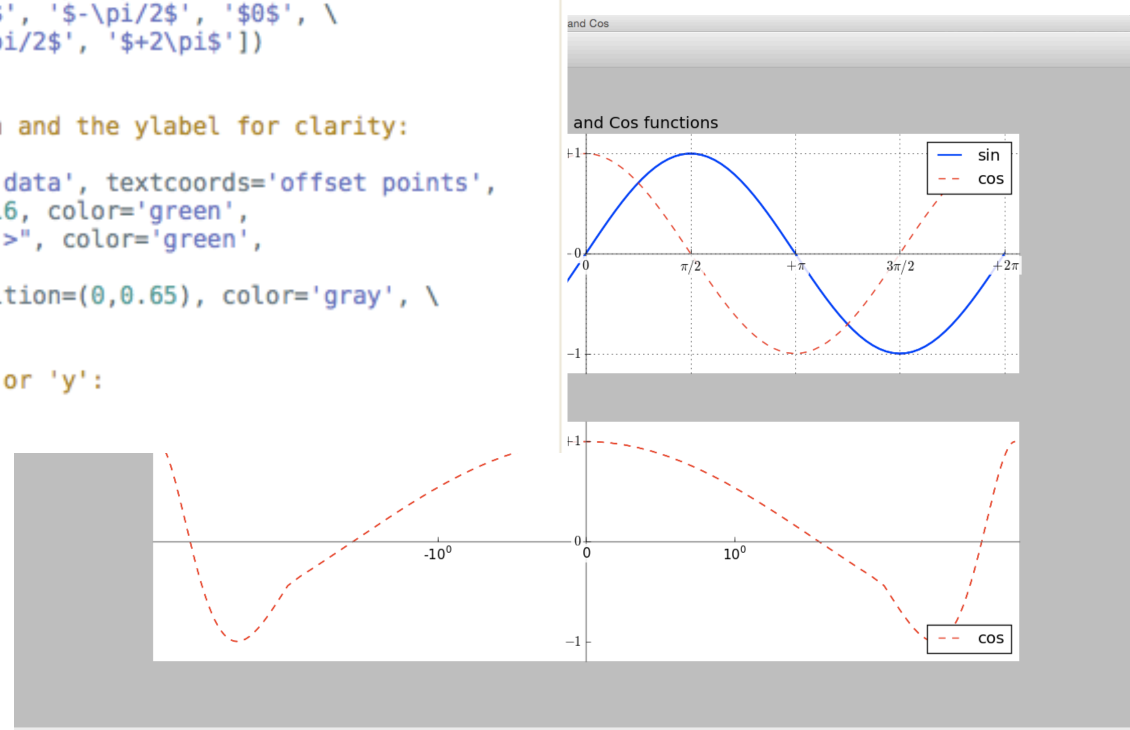
```
## adding some goodies:

# change/set the name of a figure:
fig=plb.gcf()
fig.canvas.set_window_title('Sin and Cos')

# each plot can have a Title and 'x' and 'y' labels:
plb.title('Plot of the Sin and Cos functions')
plb.xlabel('period, rd', fontsize = 9, position=(0.065,0), rotation=5, \
            color='gray', alpha=0.75)
plb.ylabel('amplitude', fontsize = 9, position=(0,0.75), color='gray', \
            alpha=0.75)

# place a grid:
plb.grid()
```



Plotting tools

- Plotting tools: *fine touches – hold, plot over, scale change*

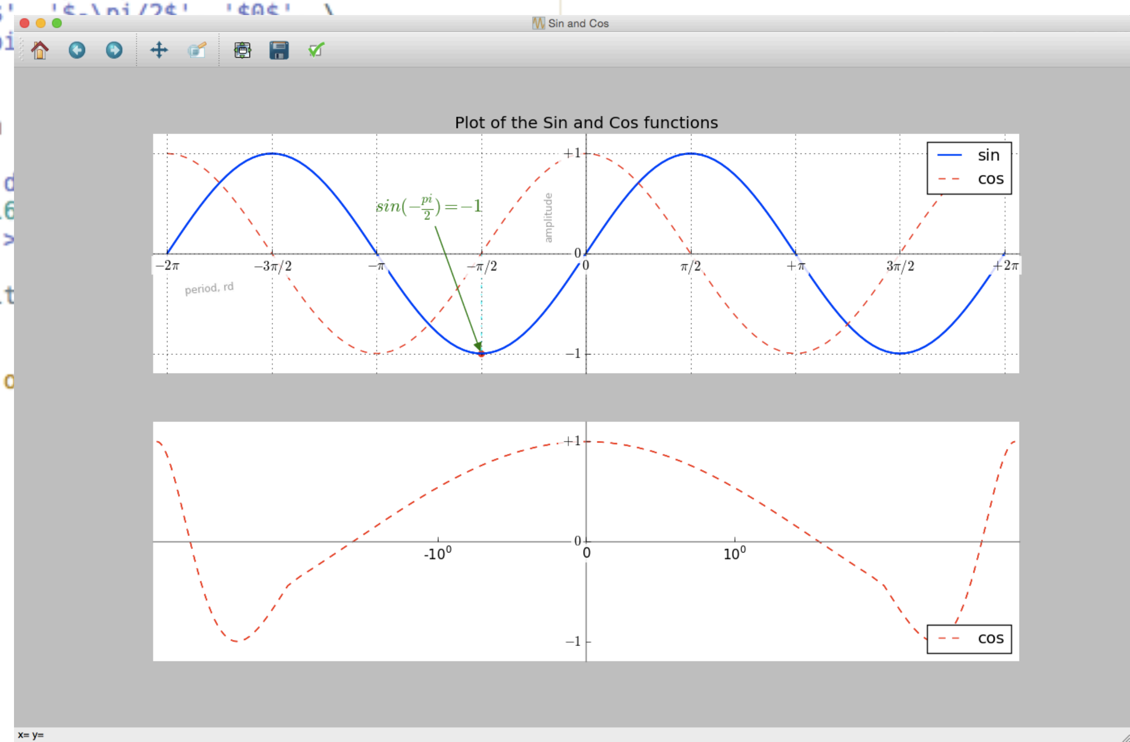
```
# hold so that another plot can be drawn on top of the current:
plb.hold(True)

# now we plot and set the x,y limits for the 'cos' function as before:
plb.plot(d, d_cos, color="red", linewidth=1, linestyle="--", label="cos")
plb.legend(loc='upper right')
plb.xlim(d_cos.min() * 6.5, d_cos.max() * 6.5)
plb.ylim(d_cos.min() * 1.2, d_cos.max() * 1.2)
plb.xticks([-plb.pi*2, -3*plb.pi/2, -plb.pi, -plb.pi/2, 0,
                plb.pi/2, plb.pi, 3*plb.pi/2, plb.pi*2],
          ['$-2\pi$', '$-3\pi/2$', '$-\pi$', '$-\pi/2$', '$0$', \
                '$\pi/2$', '$+\pi$', '$3\pi/2$', '$+2\pi$'])
plb.yticks([-1, 0, +1])

# we change the position pf the annotation and the ylabel for clarity:
plb.annotate(r'$sin(-\frac{pi}{2})=-1$',
            xy=(i, plb.sin(i)), xycoords='data', textcoords='offset points',
            xytext=(-95, +125), fontsize=16, color='green',
            arrowprops=dict(arrowstyle="-|>", color='green',
            connectionstyle="arc"))
plb.ylabel('amplitude', fontsize = 9, position=(0,0.65), color='gray', \
            alpha=0.75)

# we can change the plotting scale on 'x' or 'y':
plb.subplot(2,1,2)
plb.xscale('symlog')
```



Plotting tools

- Plotting tools: *fine touches – hold, plot over, scale change*

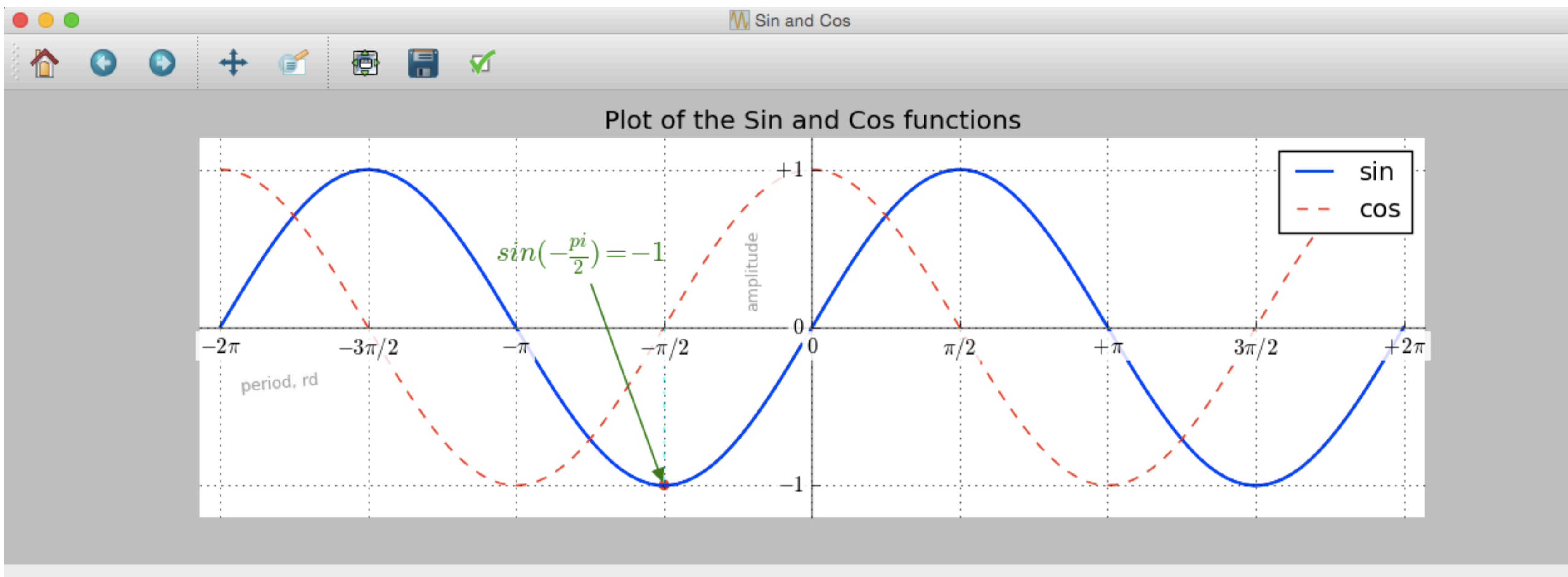
```
# hold so that another plot can be drawn on top of the current:
plb.hold(True)

# now we plot and set the x,y limits for the 'cos' function as before:
plb.plot(d, d_cos, color="red", linewidth=1, linestyle="--", label="cos")
plb.legend(loc='upper right')
plb.xlim(d_cos.min() * 6.5, d_cos.max() * 6.5)
plb.ylim(d_cos.min() * 1.2, d_cos.max() * 1.2)
plb.xticks([-plb.pi*2, -3*plb.pi/2, -plb.pi, -plb.pi/2, 0,
                plb.pi/2, plb.pi, 3*plb.pi/2, plb.pi*2],
            ['$-2\pi$', '$-3\pi/2$', '$-\pi$',
                '$\pi/2$', '$+\pi$', '$3\pi
plb.yticks([-1, 0, +1])

# we change the position pf the annotation
plb.annotate(r'$sin(-\frac{pi}{2})=-1$',
             xy=(i, plb.sin(i)), xycoords='d
             xytext=(-95, +125), fontsize=16
             arrowprops=dict(arrowstyle="-|>
             connectionstyle="arc"))
plb.ylabel('amplitude', fontsize = 9, posit
             alpha=0.75)

# we can change the plotting scale on 'x' o
plb.subplot(2,1,2)
plb.xscale('symlog')
```



Plotting tools

- Plotting tools: *fine touches – remove subplot, adjust legend & opacity*

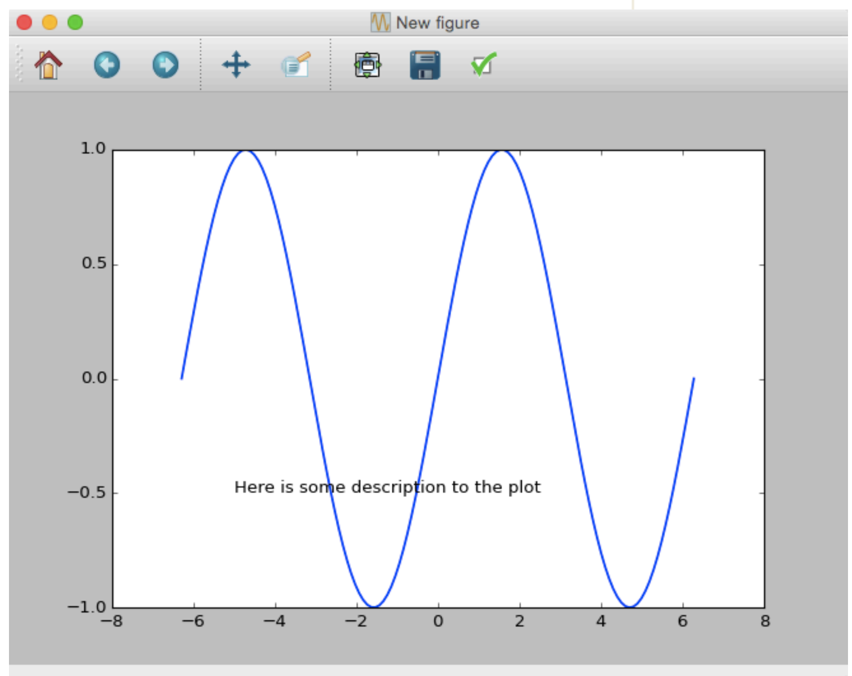
```
# to remove the second subplot at position (2,1,2) do this:
ax1.set_visible(False)
ax2.change_geometry(1,1,1)

# we need to adjust the legend:
ax2.legend(loc=1)
fig.canvas.draw()
```



Plotting tools

- Plotting tools: *other options – more figures, figure name, pause, close, text*

```
# user can create a separate figure:
plb.figure(2, dpi=65)
# closes the current figure after pausing for 5 seconds:
plb.pause(5)
plb.close()

# user can specify the name of a figure:
plb.figure('New figure')
plb.plot(d, d_sin, color="blue",  linewidth=1.5, linestyle="-", label="sin")
string = ('Here is some description to the plot')
plb.text(-5,-0.5,string)
```



Plotting tools

- Plotting tools:

  … so far we saw that:

  - when using the *figure* command, we refer to the whole graphical area
  - within the figure *subplot* can be placed in different parts of the graphical area
  - a default call to create a figure opens a figure area with default title 'Figure #'
  - figures in Python are numbered starting from 1 (not from 0) just like in Matlab
  - there are several optional parameters that define how a figure should appear

Option	Default value	Meaning
num	1	number of figure
dpi	figure.dpi	resolution in dots per inch
figsize	figure.figsize	figure size (width, height), in inches
frameon	TRUE	to draw figure frame or not
facecolor	figure.facecolor	background color of the drawing
edgecolor	figure.edgecolor	edge color around the drawing background

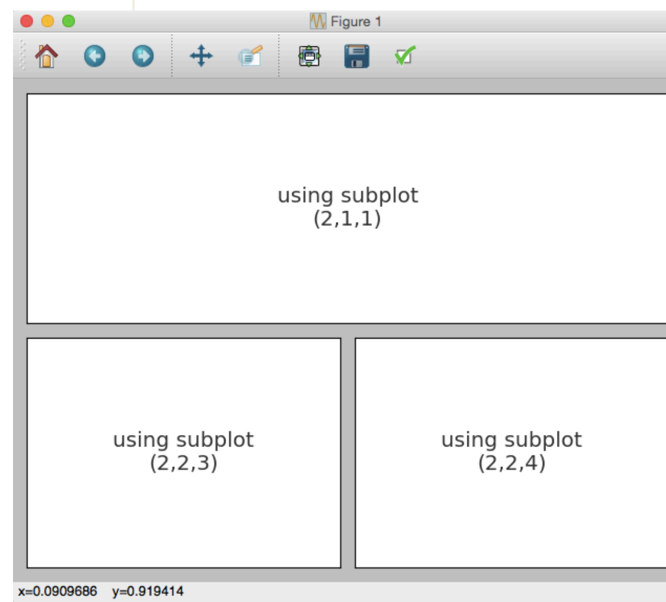
Plotting tools

- Plotting tools:

  … so far we saw that:

  – *subplot* places a plot in a regular grid, within the *figure* space

```
# subplot example:
plb.subplot(2, 1, 1)
plb.xticks(()), plb.yticks(())
plb.text(0.5, 0.5, 'using subplot\n(2,1,1)', ha='center', va='center',
        size=18, alpha=.8)

plb.subplot(2, 2, 3)
plb.xticks(()), plb.yticks(())
plb.text(0.5, 0.5, 'using subplot\n(2,2,3)', ha='center', va='center',
        size=18, alpha=.8)

plb.subplot(2, 2, 4)
plb.xticks(()), plb.yticks(())
plb.text(0.5, 0.5, 'using subplot\n(2,2,4)', ha='center', va='center',
        size=18, alpha=.8)

plb.tight_layout() # makes the sqares tighter to one another
plb.show()
```



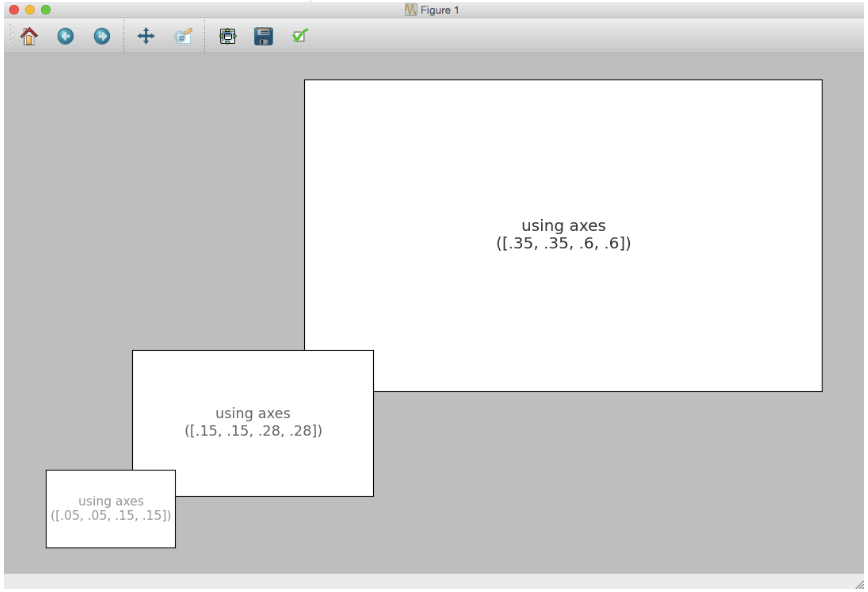
Plotting tools

- Plotting tools:

  … so far we saw that:

  – *axes* provides a free placement of the plot inside of the *figure*

```
# axes example:
plb.axes([.35, .35, .6, .6])
plb.xticks(()), plb.yticks(())
plb.text(.5, .5, 'using axes\n([.35, .35, .6, .6])', ha='center', va='center',
        size=18, alpha=.8)

plb.axes([.15, .15, .28, .28])
plb.xticks(()), plb.yticks(())
plb.text(.5, .5, 'using axes\n([.15, .15, .28, .28])', ha=
        size=16, alpha=.6)

plb.axes([.05, .05, .15, .15])
plb.xticks(()), plb.yticks(())
plb.text(.5, .5, 'using axes\n([.05, .05, .15, .15])', ha=
        size=14, alpha=.4)

plb.show()
```



Plotting tools

- Plotting tools:

  … so far we saw that:

  - when in the call none of the options are used, then *figure()* is called that makes a default *subplot* at position (111)
  - when a call is made to *plot*, matplotlib calls *gca()* and gets the <u>c</u>urrent <u>a</u>xes
  - *gca()* calls *gcf()* to provide the current figure
  - tick locators are several types and can be set to the specific needs: *null, linear, log, etc.*
  - creating figures and axes <u>implicitly</u> is nice and quick, but offers limited usage
  - <u>explicit</u> figure reference will provide more control over the display, while taking full advantage of figure, subplot, and axes

Homework #3

Part 1 – create your data:

1. Include a section line with your name
2. Work only with these imports:

   from numpy import matrix, array, random, min, max

   import pylab as plb (… or use matplotlib)
3. Cerate a list A of 600 random numbers bound between (0:10)
4. Create an array B with with 500 elements bound in the range [-3*pi:2*pi]
5. Using *if*, *for* or *while*, create a function that overwrites every element in A that falls outside of the interval [2:9), and overwrite that element with the average between the smallest and largest element in A
6. Normalize each list element to be bound between [0:0.1]
7. Return the result from the function to C
8. Cast C as an array
9. Add C to B (think of C as noise) and record the result in D … (watch out: C is of different length. Truncate it)

Part 2 - plotting:

10. Create a figure, give it a title and specify your own size and dpi
11. Plot the sin of D, in the (2,1,1) location of the figure
12. Overlay a plot of cos using D, with different color, thickness and type of line
13. Create some space on top and bottom of the plot (on the y axis) and show the grid
14. Specify the following: title, Y-axis label and legend to fit in the best way
15. Plot the tan of D, in location (2,1,2) with grid showing, X-axis label, Y-axis label and legend on top right
16. Organize your code: use each line from this HW as a comment line before coding each step
17. Save these steps in a .py file and email it to me before next class. I will run it!