

# 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, IPython and the editor HW1
- **Basic language specifics 1/2**
- Basic arithmetic operations, assignment operators, data types, containers
- Control flow (if/elif/else)
- Conditional expressions
- Iterative 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. *project discussion*

# HW2 – discussion

- HW2 – discussion

```
1 # 1. Include a section line with your name:
2 ## HW 2: Alexander Iliev
3 from numpy import matrix, array, random, min, max
4
5 # 2. Create Matrix A with size (3,5) containing random numbers:
6 A = random.random(15)
7 A = A.reshape(3,5)
8 A = matrix(A)
9
10 # 3. Find the size and length of Matrix A:
11 A.size
12 len(A)
13
14 # 4. Resize (crop) Matrix A to size (3,4):
15 A = A[0:3,0:4]
16
17 # 5. Find the transpose of Matrix A and record it as Matrix B:
18 B = A.T
19
20 # 6. Find the minimum value in column 1 of each row of Matrix B:
21 B[:,1].min()
22
23 # 7. Find the minimum and maximum values for the entire Matrix A:
24 A.min()
25 A.max()
26
27 # 8. Create Vector X (an array) with 4 random numbers:
28 X = array([random.random(4)])
29
30 # 9. Create a function and pass Vector X and Matrix A in it:
31 def function_HW2(a,b):
32     return a*b.T
33
34 # 10. In the new function multiply Vector X with Matrix A
35 # and return the result back to the main program (result D):
36 D = function_HW2(X,A)
37
38 # 11. Create a complex number Z with absolute and real parts ~= 0:
39 Z = 6+5j
40
41 # 12. Show its real and imaginary parts:
42 Z.real
43 Z.imag
44 abs(Z)
45
46 # 13. Multiply result D with the absolute value of Z and record it to C:
47 C = D*abs(Z)
48
49 # 14. Convert Matrix B from a matrix to a string and overwrite B:
50 B = str(B)
51
52 # 15. Display a text on the screen: 'I am done with HW2',
53 # but pass 'HW2' as a string variable:
54 print('%s is done with HW2' % 'Alex')
```

# Random options in NumPy

- Random options in NumPy

```
In [55]: np.random.rand(3,2)
Out[55]:
array([[0.83252202, 0.38362844],
       [0.63309619, 0.29230593],
       [0.95616478, 0.07331045]])
```

Random values in a given shape.

Create an array of the given shape and populate it with random samples from a uniform distribution over `[0, 1)`.

```
In [56]: np.random.randn(3,2)
Out[56]:
array([[ -1.326648  , -0.43371855],
       [ 0.76498162,  0.21129565],
       [ 0.06951502,  0.24104293]])
```

`randn(d0, d1, ..., dn)`

Return a sample (or samples) from the "standard normal" distribution.

```
In [57]: np.random.random(3)
Out[57]: array([0.91596981, 0.60502728, 0.11632779])
```

Return random floats in the half-open interval `[0.0, 1.0)`.

```
In [58]: np.random.randint(30,45)
Out[58]: 38
```

`randint(low, high=None, size=None, dtype='l')`

Return random integers from `low` (inclusive) to `high` (exclusive).

# 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?

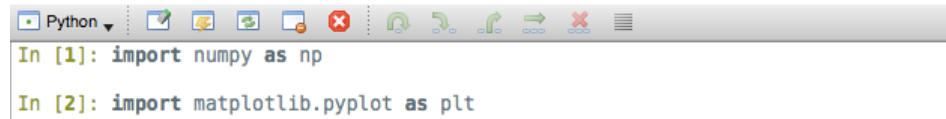
- What is Matplotlib?
  - Matplotlib also supports multiple **non-interactive backend** systems like: postscript, PDF, SVG, antigrain geometry and Cairo
  - Matplotlib has several **dependencies**, one of which is **NumPy**, but Scipy is not
  - Matplotlib plots can be:
    - used in publishing material
    - 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:



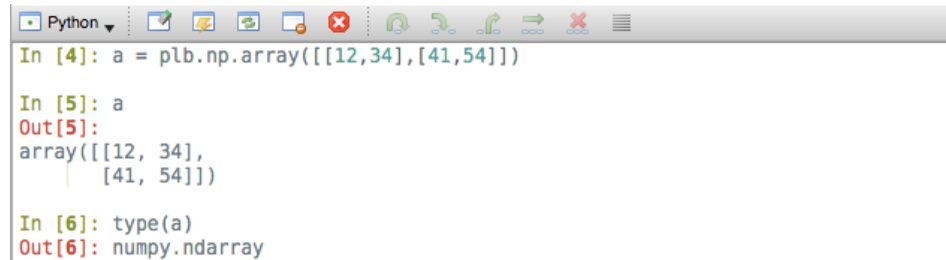
```
Python  
In [1]: import numpy as np  
In [2]: import matplotlib.pyplot as plt
```

or



```
Python  
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:



```
Python  
In [4]: a = plb.numpy.array([[12,34],[41,54]])  
  
In [5]: a  
Out[5]:  
array([[12, 34],  
       [41, 54]])  
  
In [6]: type(a)  
Out[6]: numpy.ndarray
```

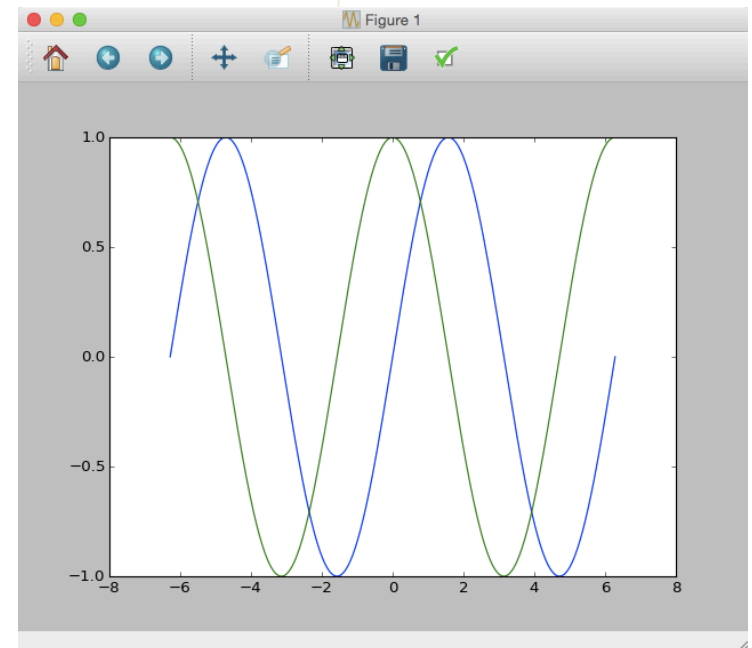
so that *creating an array via pylab* is the same as creating it as if you imported NumPy



# Basic plotting

- Basic plotting: *comparison*

```
1  ## Basic plotting using numpy and matplotlib.pyplot:
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  # lets create the array 'a' with 512 points in the range [-2*pi:2*pi]:
6  a = np.linspace(-np.pi*2, np.pi*2, 512, endpoint=True)
7
8  # the 'sin' and 'cos' functions have the same number of points (512):
9  b_sin, c_cos = np.sin(a), np.cos(a)
10
11 # lets plot the results of the two functions above:
12 plt.plot(a, b_sin)
13 plt.plot(a, c_cos)
14 plt.show()
```

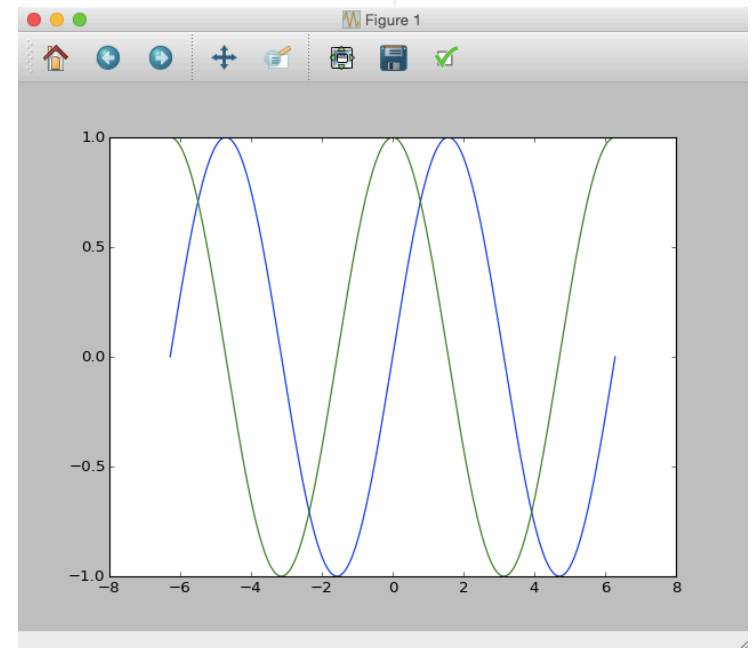


# Basic plotting

- Basic plotting: *comparison*

```
16 ## Basic plotting using pylab:
17 import pylab as plb
18
19 # lets create the array 'a' with 512 points in the range [-2*pi:2*pi]:
20 a = plb.linspace(-plb.pi*2, plb.pi*2, 512, endpoint=True)
21
22 # the 'sin' and 'cos' functions have the same number of points (512):
23 b_sin, c_cos = plb.sin(a), plb.cos(a)
24
25 # lets plot the results of the two functions above:
26 plb.plot(a, b_sin)
27 plb.plot(a, c_cos)
28 plb.show()
```

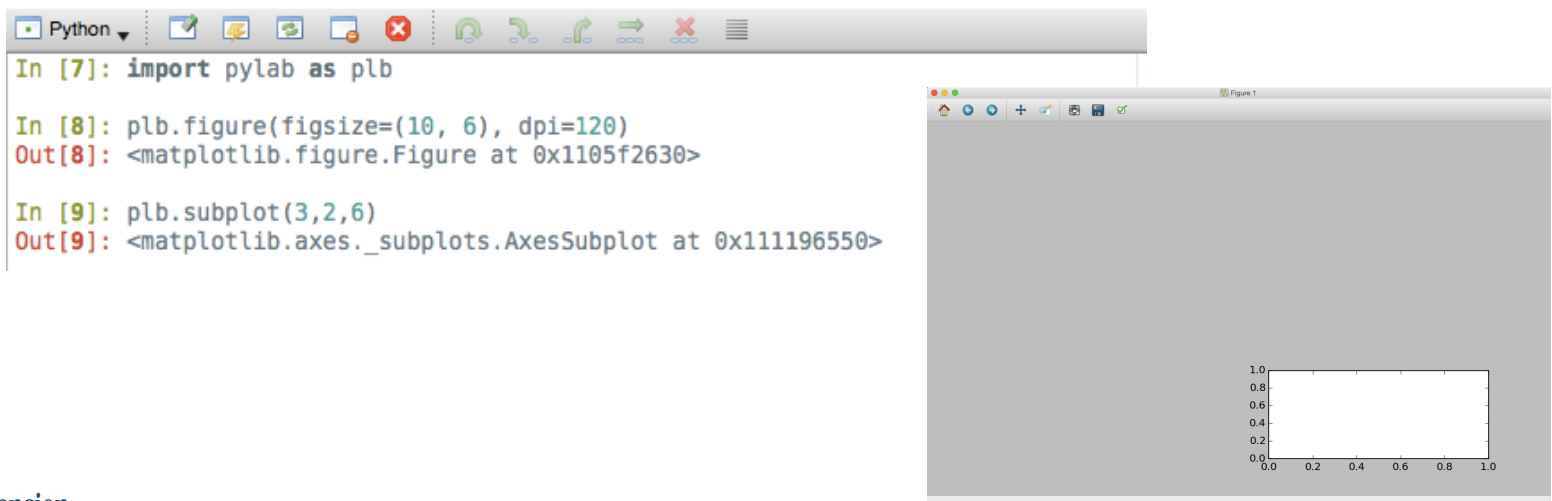
both cases produce the same result ----->



# 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):  
`plt.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':  
`plt.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'

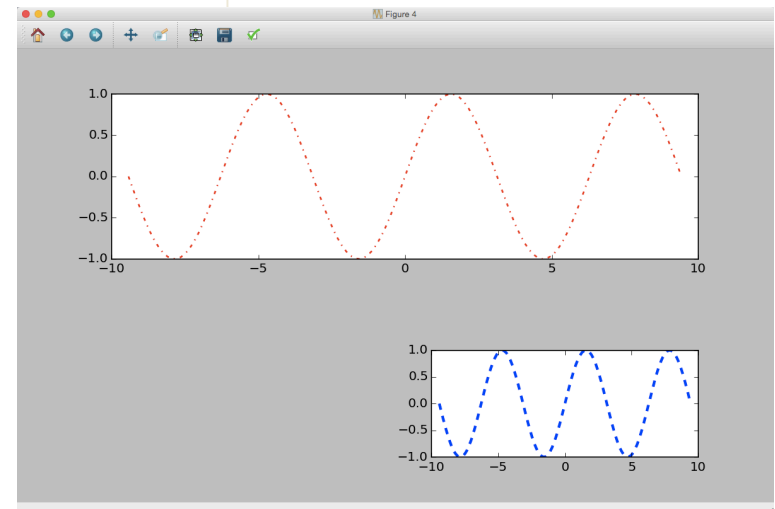
Example:



# Basic plotting tools

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

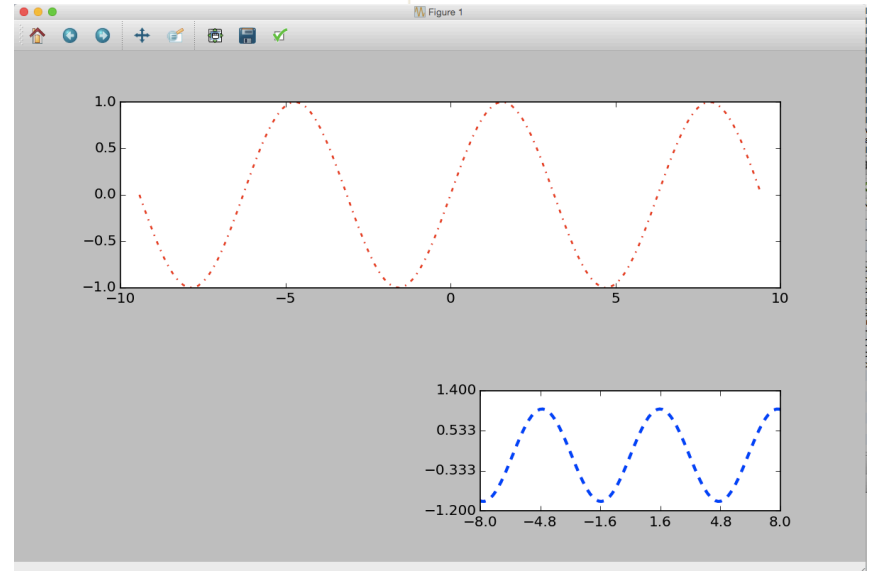
```
30 ## Basic plotting tools:
31 import pylab as plb
32 plb.figure(figsize=(10, 6), dpi=120)
33
34 # we create an array 'd' with 128 points in the range [-3*pi:3*pi]:
35 d = plb.linspace(-plb.pi*3, plb.pi*3, 128, endpoint=True)
36
37 # now we create the 'sin' and 'cos' functions from 'd' with 128 points each:
38 d_sin = plb.sin(d)
39 d_cos = plb.cos(d)
40
41 # plot 'sin' using a green dash-dotted line of width 1.5px in area (2,1,1):
42 plb.subplot(2,1,1)
43 plb.plot(d, d_sin, color="red", linewidth=1.5, linestyle="-.")
44
45 # plot 'cos' using a blue dashed line of width 1.5px in area (3,2,6):
46 plb.subplot(3,2,6)
47 plb.plot(d, d_cos, color="blue", linewidth=2.5, linestyle="--")
```



# Plotting tools

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

```
49 # we need to set the 'x' limits:
50 plb.xlim(-8.0, 8.0)
51 # then plot 'x' ticks:
52 plb.xticks(plb.linspace(-8, 8, 6, endpoint=True))
53
54 # now we set the 'y' limits:
55 plb.ylim(-1.2, 1.4)
56 # we set the 'y' ticks:
57 plb.yticks(plb.linspace(-1.2, 1.4, 4, endpoint=True))
58
59 # show the result on screen:
60 plb.show()
61
62 # we can now save the figure using 64 dots per inch:
63 plb.savefig("lecture_5.png", dpi=64)
```

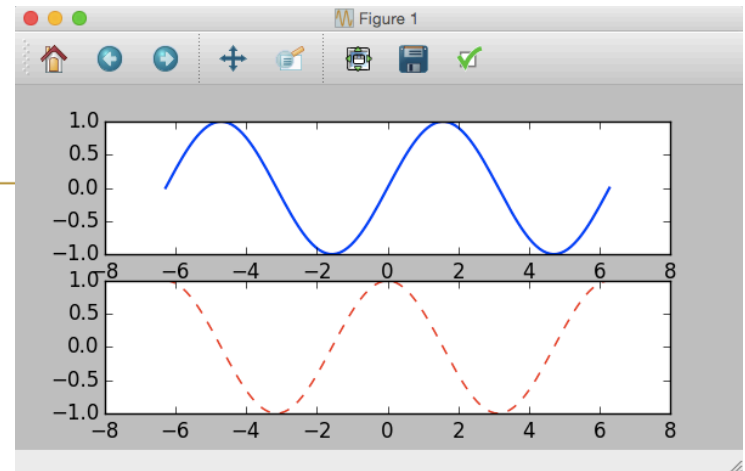


# Plotting tools

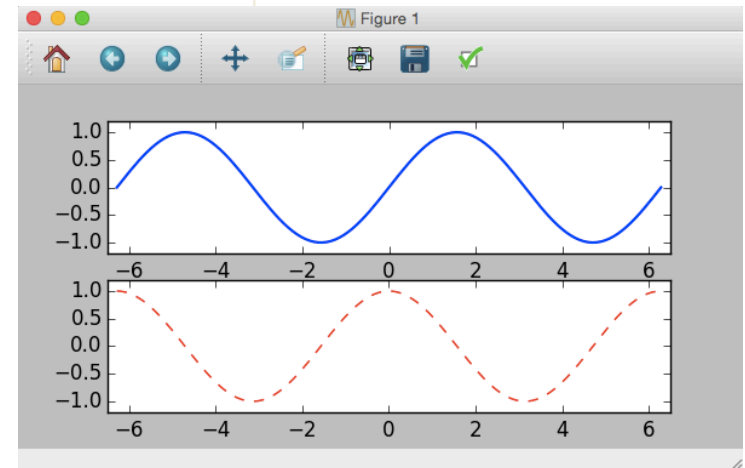
- Plotting tools: *changing plot limits*

```
65 ## Changing plot limits:
66 import pylab as plb
67
68 plb.figure(figsize=(6, 3), dpi=100)
69 d = plb.linspace(-plb.pi*2, plb.pi*2, 128, endpoint=True)
70 d_sin = plb.sin(d)
71 d_cos = plb.cos(d)
72
73 # we now set the x,y limits for the 'sin' function:
74 plb.subplot(2,1,1)
75 plb.plot(d, d_sin, color="blue", linewidth=1.5, linestyle="-")
76 plb.xlim(d_sin.min() * 6.5, d_sin.max() * 6.5)
77 plb.ylim(d_sin.min() * 1.2, d_sin.max() * 1.2)
78
79 # below we set the x,y limits for the 'cos' function:
80 plb.subplot(2,1,2)
81 plb.plot(d, d_cos, color="red", linewidth=1, linestyle="--")
82 plb.xlim(d_cos.min() * 6.5, d_cos.max() * 6.5)
83 plb.ylim(d_cos.min() * 1.2, d_cos.max() * 1.2)
```

before



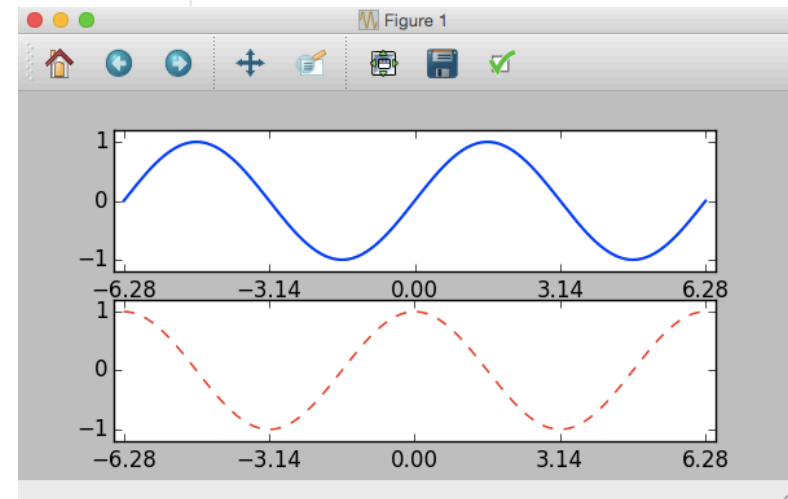
after



# Plotting tools

- Plotting tools: *editing ticks*

```
65 ## Changing plot limits:
66 import pylab as plb
67
68 plb.figure(figsize=(6, 3), dpi=100)
69 d = plb.linspace(-plb.pi*2, plb.pi*2, 128, endpoint=True)
70 d_sin = plb.sin(d)
71 d_cos = plb.cos(d)
72
73 # we now set the x,y limits for the 'sin' function:
74 plb.subplot(2,1,1)
75 plb.plot(d, d_sin, color="blue", linewidth=1.5, linestyle="-")
76 plb.xlim(d_sin.min() * 6.5, d_sin.max() * 6.5)
77 plb.ylim(d_sin.min() * 1.2, d_sin.max() * 1.2)
78 plb.xticks([-plb.pi*2, -plb.pi, 0, plb.pi, plb.pi*2]) #<-----
79 plb.yticks([-1, 0, +1])
80
81 # below we set the x,y limits for the 'cos' function:
82 plb.subplot(2,1,2)
83 plb.plot(d, d_cos, color="red", linewidth=1, linestyle="--")
84 plb.xlim(d_cos.min() * 6.5, d_cos.max() * 6.5)
85 plb.ylim(d_cos.min() * 1.2, d_cos.max() * 1.2)
86 plb.xticks([-plb.pi*2, -plb.pi, 0, plb.pi, plb.pi*2]) #<-----
87 plb.yticks([-1, 0, +1])
```



# Plotting tools

- Plotting tools: *adding tick labels*

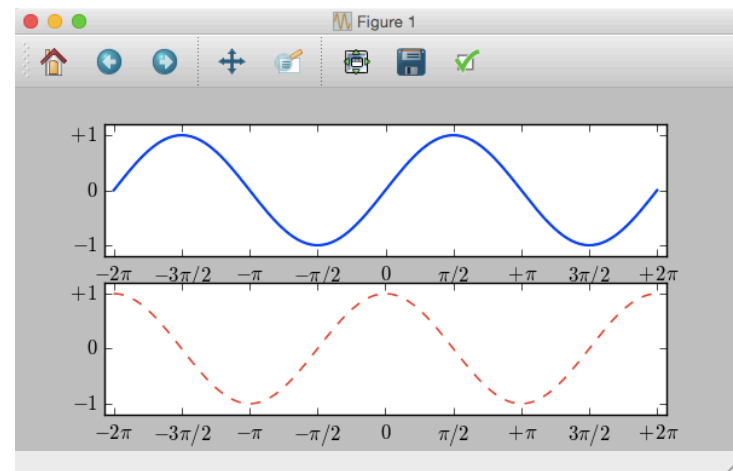
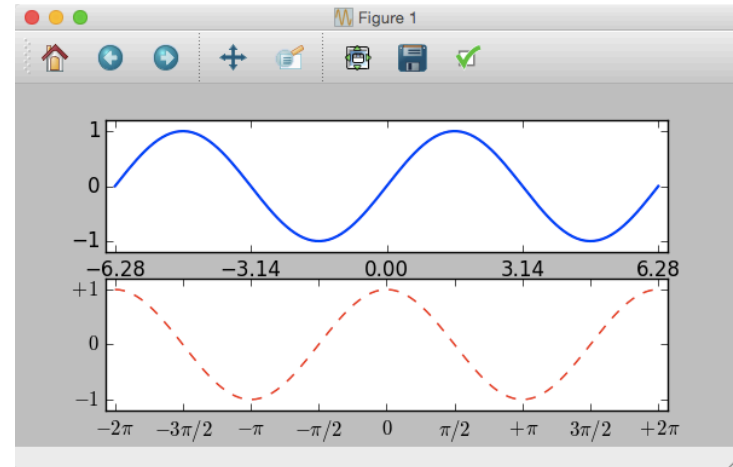
Now that we set the ticks correctly, we need to be a bit more explicit about what they represent, so we add the following code:

```
89 # adding x,y tick labels for plot (2,1,2):
90 plb.xticks([-plb.pi*2, -3*plb.pi/2, -plb.pi, -plb.pi/2, 0,
91             plb.pi/2, plb.pi, 3*plb.pi/2, plb.pi*2],
92             ['$-2\pi$', '$-3\pi/2$', '$-\pi$', '$-\pi/2$', '$0$', '\
93             '\pi/2$', '$+\pi$', '$3\pi/2$', '$+2\pi$'])
94 plb.yticks([-1, 0, +1],
95             ['$-1$', '$0$', '$+1$'])
```

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

```
97 # now adding x,y tick labels for plot (2,1,1):
98 plb.subplot(2,1,1)
99 plb.xticks([-plb.pi*2, -3*plb.pi/2, -plb.pi, -plb.pi/2, 0,
100            plb.pi/2, plb.pi, 3*plb.pi/2, plb.pi*2],
101            ['$-2\pi$', '$-3\pi/2$', '$-\pi$', '$-\pi/2$', '$0$', '\
102            '\pi/2$', '$+\pi$', '$3\pi/2$', '$+2\pi$'])
103 plb.yticks([-1, 0, +1],
104            ['$-1$', '$0$', '$+1$'])
```

line  
split

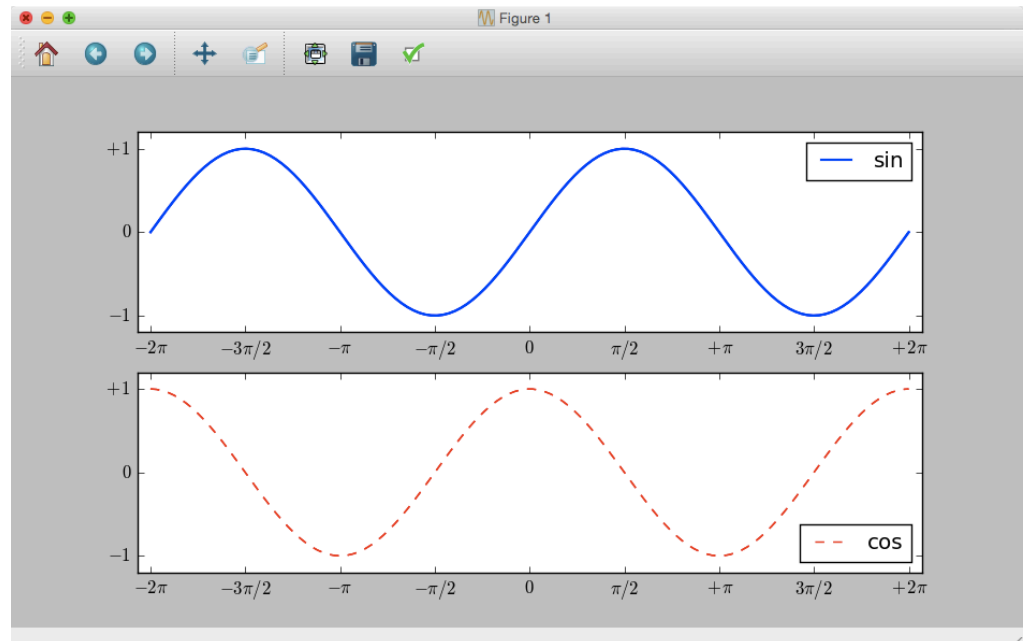




# Plotting tools

- Plotting tools: *adding a legend*

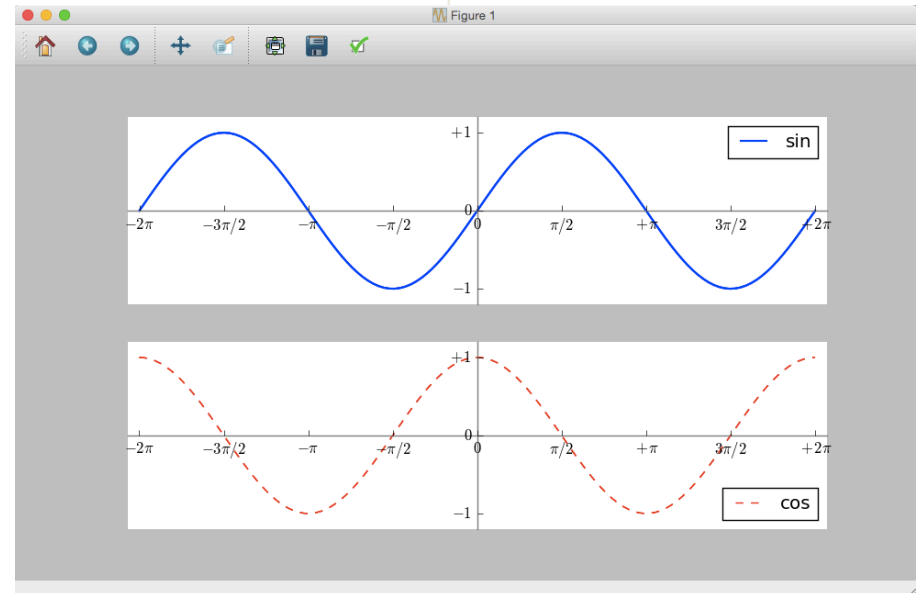
```
106 # adding a legend clarifying the plots:
107 plb.subplot(2,1,1)
108 plb.plot(d, d_sin, color="blue", linewidth=1.5, linestyle="-", label="sin")
109 plb.legend(loc='upper right')
110 plb.subplot(2,1,2)
111 plb.plot(d, d_cos, color="red", linewidth=1, linestyle="--", label="cos")
112 plb.legend(loc='lower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



# Plotting tools

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

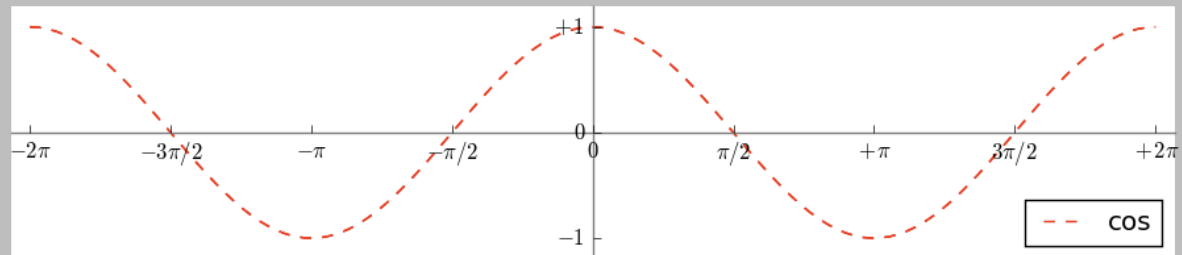
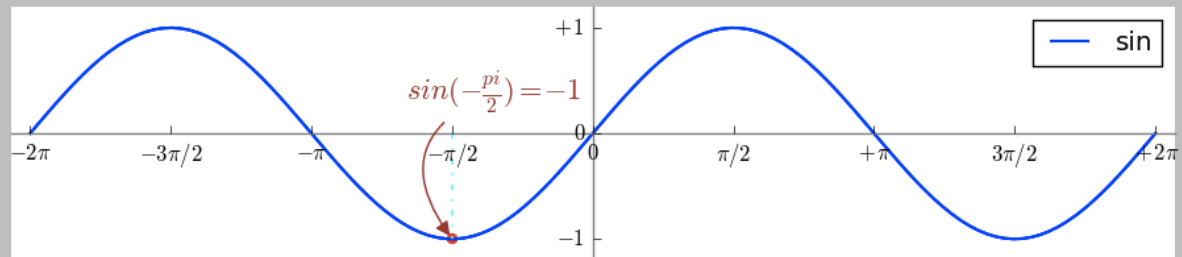
```
114 # setting the axis:
115 ax1 = plt.gca() # gca - 'get current axis'
116 ax1.spines['top'].set_color('none') # to get rid of the black border line
117 ax1.spines['bottom'].set_color('none') # to get rid of the black border line
118 ax1.spines['left'].set_color('none') # to get rid of the black border line
119 ax1.spines['right'].set_color('none') # to get rid of the black border line
120 ax1.xaxis.set_ticks_position('bottom')
121 ax1.spines['bottom'].set_position(('data',0))
122 ax1.spines['bottom'].set_color('gray')
123 ax1.yaxis.set_ticks_position('left')
124 ax1.spines['left'].set_position(('data',0))
125 ax1.spines['left'].set_color('gray')
126
127 plt.subplot(2,1,1)
128 ax2 = plt.gca() # gca - 'get current axis'
129 ax2.spines['top'].set_color('none')
130 ax2.spines['bottom'].set_color('none')
131 ax2.spines['left'].set_color('none')
132 ax2.spines['right'].set_color('none')
133 ax2.xaxis.set_ticks_position('bottom')
134 ax2.spines['bottom'].set_position(('data',0))
135 ax2.spines['bottom'].set_color('gray')
136 ax2.yaxis.set_ticks_position('left')
137 ax2.spines['left'].set_position(('data',0))
138 ax2.spines['left'].set_color('gray')
```



# Plotting tools

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

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

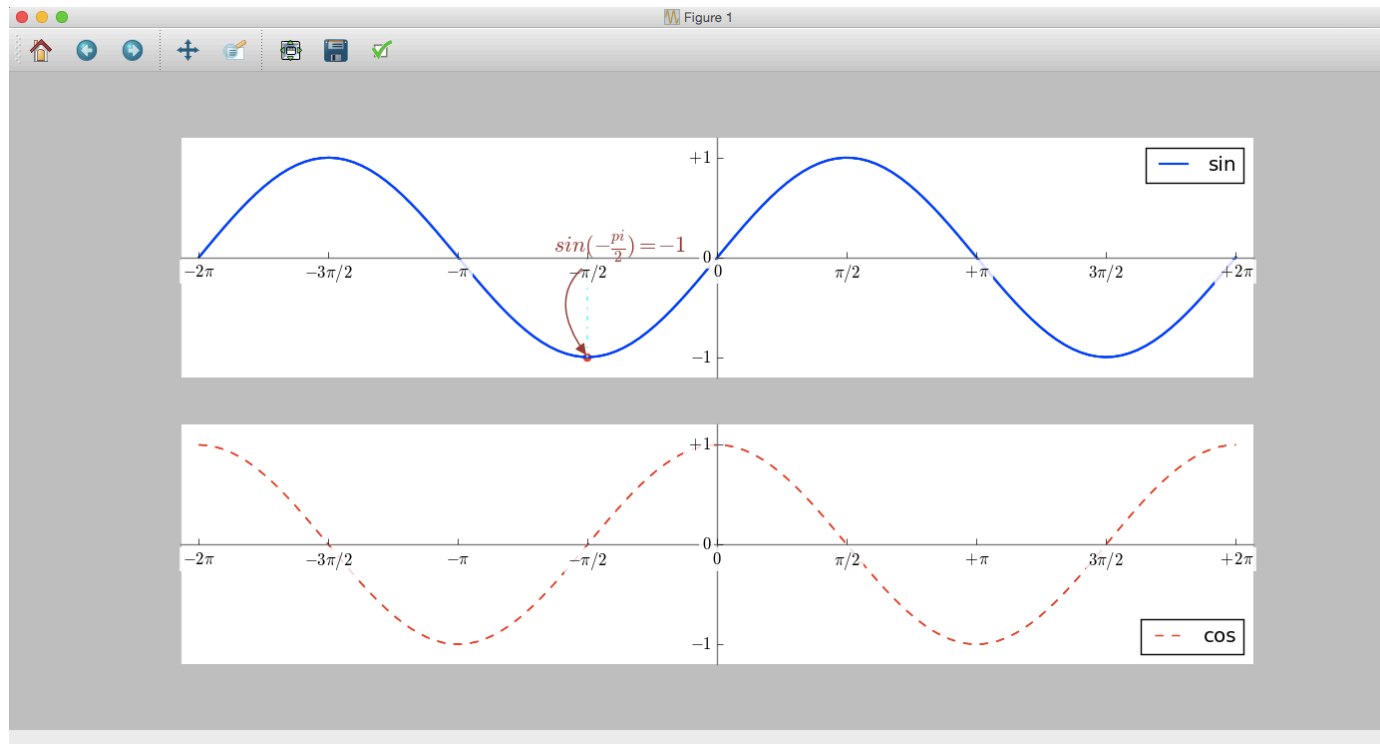


# Plotting tools

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

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

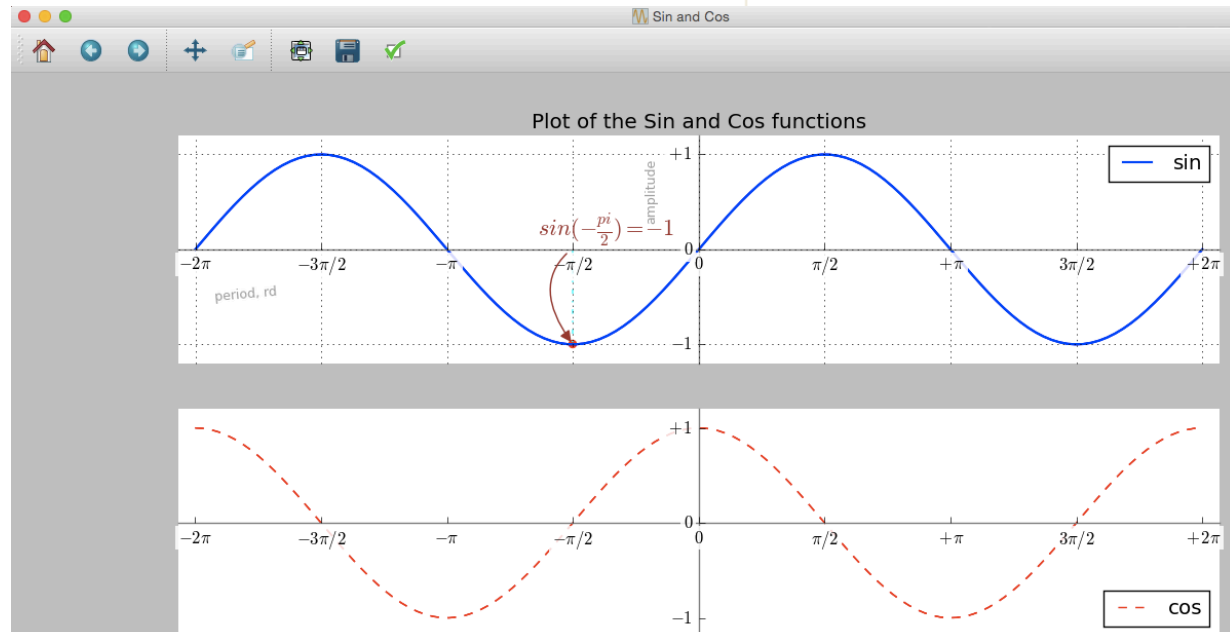
our tick labels are obscured by the plot lines running over them, so we need to make them more clear and visible



# Plotting tools

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

```
156 ## adding some goodies:
157
158 # change/set the name of a figure:
159 fig=plt.gcf()
160 fig.canvas.set_window_title('Sin and Cos')
161
162 # each plot can have a Title and 'x' and 'y' labels:
163 plt.title('Plot of the Sin and Cos functions')
164 plt.xlabel('period, rd', fontsize = 9, position=(0.065,0), rotation=5, \
165           color='gray', alpha=0.75)
166 plt.ylabel('amplitude', fontsize = 9, position=(0,0.75), color='gray', \
167           alpha=0.75)
168
169 # place a grid:
170 plt.grid()
```



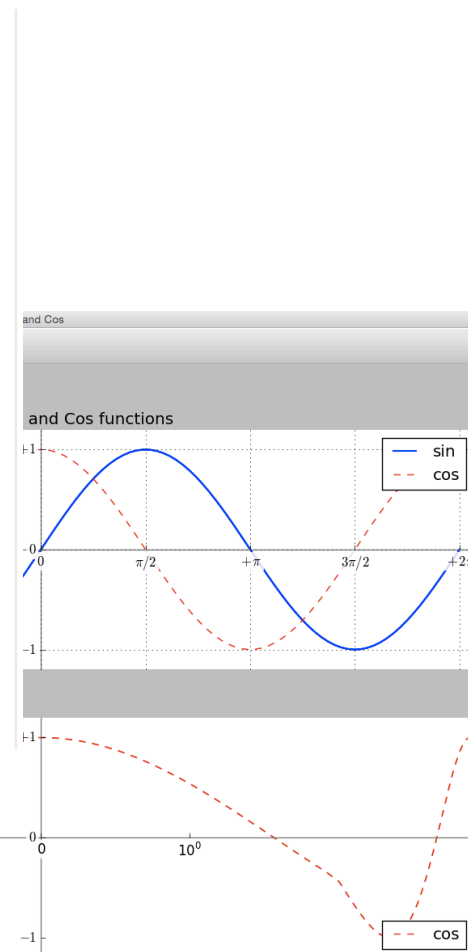
# Plotting tools

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

```

172 # hold so that another plot can be drawn on top of the current:
173 plb.hold(True)
174
175 # now we plot and set the x,y limits for the 'cos' function as before:
176 plb.plot(d, d_cos, color="red", linewidth=1, linestyle="--", label="cos")
177 plb.legend(loc='upper right')
178 plb.xlim(d_cos.min() * 6.5, d_cos.max() * 6.5)
179 plb.ylim(d_cos.min() * 1.2, d_cos.max() * 1.2)
180 plb.xticks([-plb.pi*2, -3*plb.pi/2, -plb.pi, -plb.pi/2, 0,
181            plb.pi/2, plb.pi, 3*plb.pi/2, plb.pi*2],
182            ['$-2\pi$', '$-3\pi/2$', '$-\pi$', '$-\pi/2$', '$0$', \
183            '$\pi/2$', '$+\pi$', '$3\pi/2$', '$+2\pi$'])
184 plb.yticks([-1, 0, +1])
185
186 # we change the position pf the annotation and the ylabel for clarity:
187 plb.annotate(r'$\sin(-\frac{\pi}{2})=-1$',
188            xy=(i, plb.sin(i)), xycoords='data', textcoords='offset points',
189            xytext=(-95, +125), fontsize=16, color='green',
190            arrowprops=dict(arrowstyle="->", color='green',
191            connectionstyle="arc"))
192 plb.ylabel('amplitude', fontsize = 9, position=(0,0.65), color='gray', \
193            alpha=0.75)
194
195 # we can change the plotting scale on 'x' or 'y':
196 plb.subplot(2,1,2)
197 plb.xscale('symlog')

```



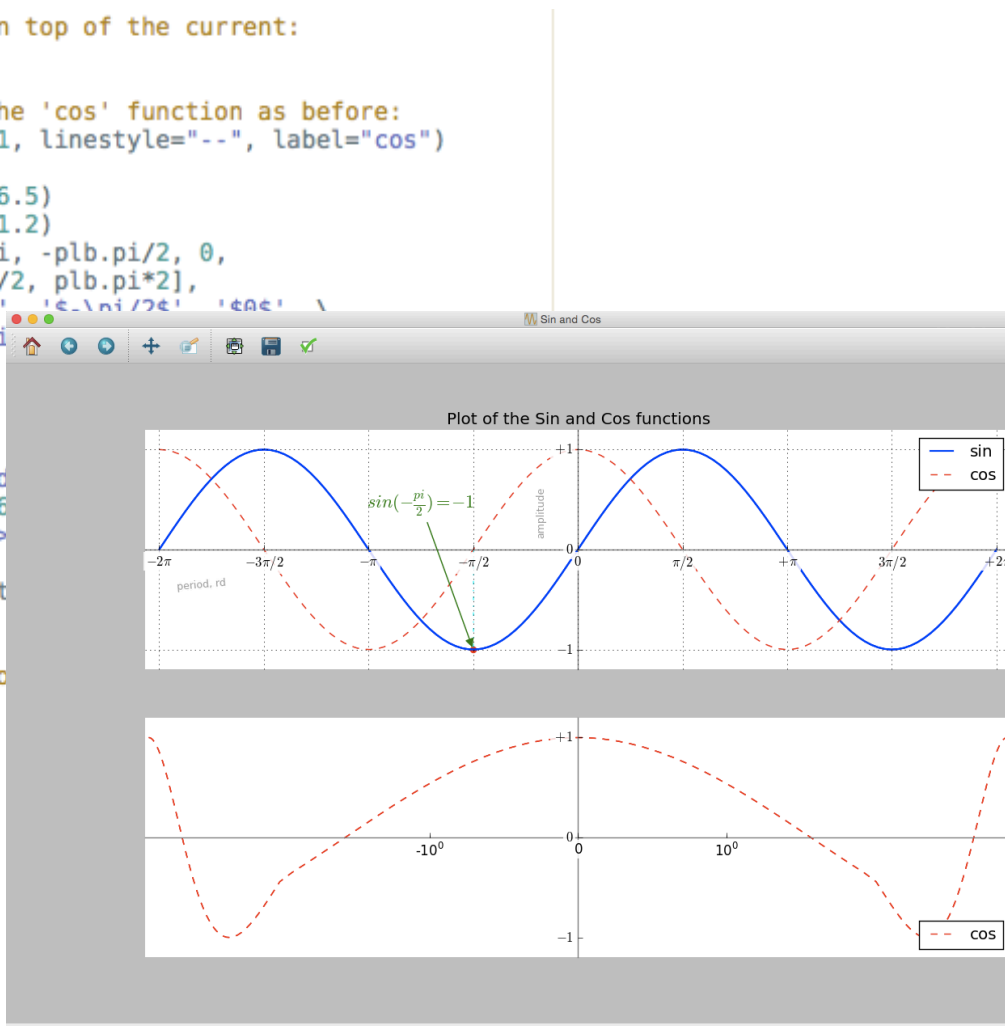
# Plotting tools

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

```

172 # hold so that another plot can be drawn on top of the current:
173 plb.hold(True)
174
175 # now we plot and set the x,y limits for the 'cos' function as before:
176 plb.plot(d, d_cos, color="red", linewidth=1, linestyle="--", label="cos")
177 plb.legend(loc='upper right')
178 plb.xlim(d_cos.min() * 6.5, d_cos.max() * 6.5)
179 plb.ylim(d_cos.min() * 1.2, d_cos.max() * 1.2)
180 plb.xticks([-plb.pi*2, -3*plb.pi/2, -plb.pi, -plb.pi/2, 0,
181            plb.pi/2, plb.pi, 3*plb.pi/2, plb.pi*2],
182            ['$-2\pi$', '$-3\pi/2$', '$-\pi$', '$-\pi/2$', '$0$',
183            '$\pi/2$', '$+\pi$', '$3\pi/2$', '$2\pi$'])
184 plb.yticks([-1, 0, +1])
185
186 # we change the position pf the annotation
187 plb.annotate(r'$\sin(-\frac{\pi}{2})=-1$',
188            xy=(i, plb.sin(i)), xycoords='data',
189            xytext=(-95, +125), fontsize=16,
190            arrowprops=dict(arrowstyle="|>",
191            connectionstyle="arc"))
192 plb.ylabel('amplitude', fontsize = 9, position =
193            alpha=0.75)
194
195 # we can change the plotting scale on 'x' of the
196 plb.subplot(2,1,2)
197 plb.xscale('symlog')

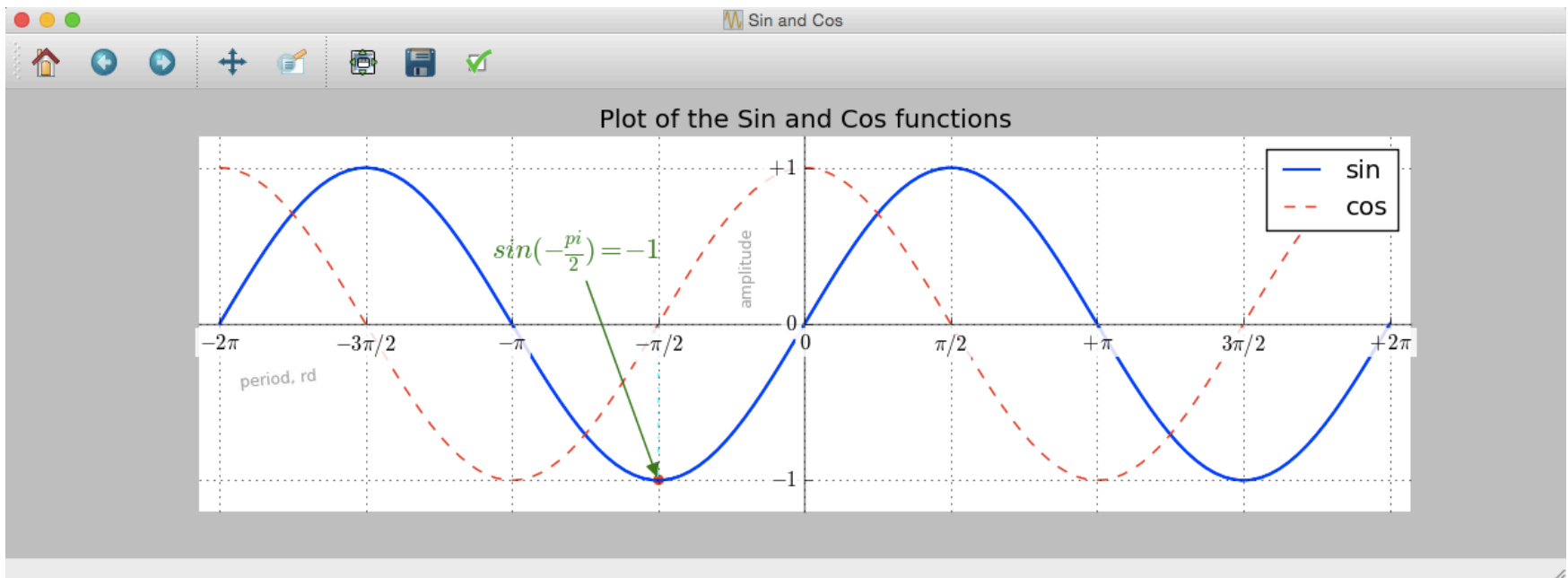
```



# Plotting tools

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

```
199 # to remove the second subplot at position (2,1,2) do this:
200 ax1.set_visible(False)
201 ax2.change_geometry(1,1,1)
202
203 # we need to adjust the legend:
204 ax2.legend(loc=1)
205 fig.canvas.draw()
```

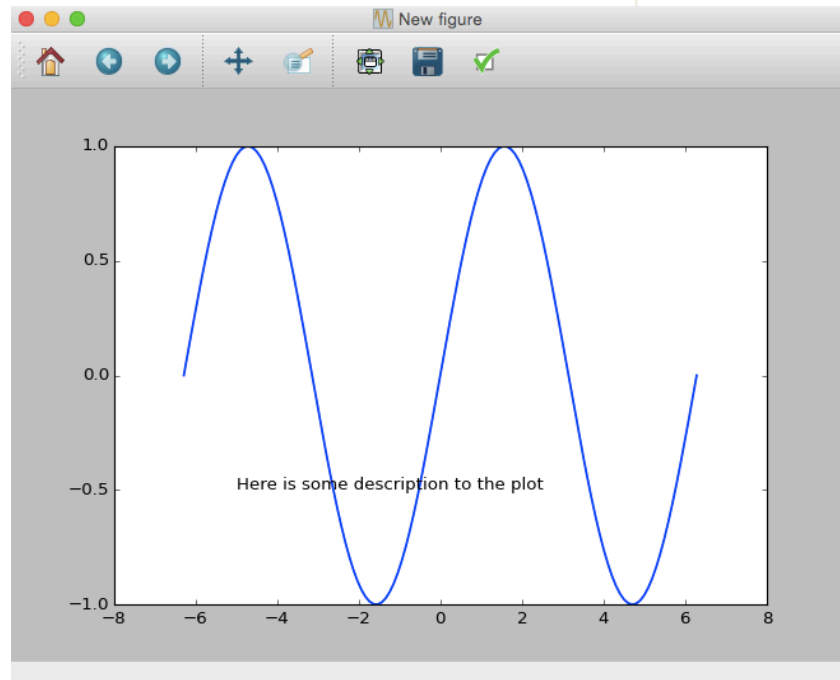




# Plotting tools

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

```
194 # user can create a separate figure:
195 plb.figure(2, dpi=65)
196 # closes the current figure after pausing for 5 seconds:
197 plb.pause(5)
198 plb.close()
199
200 # user can specify the name of a figure:
201 plb.figure('New figure')
202 plb.plot(d, d_sin, color="blue", linewidth=1.5, linestyle="-", label="sin")
203 string = ('Here is some description to the plot')
204 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 |

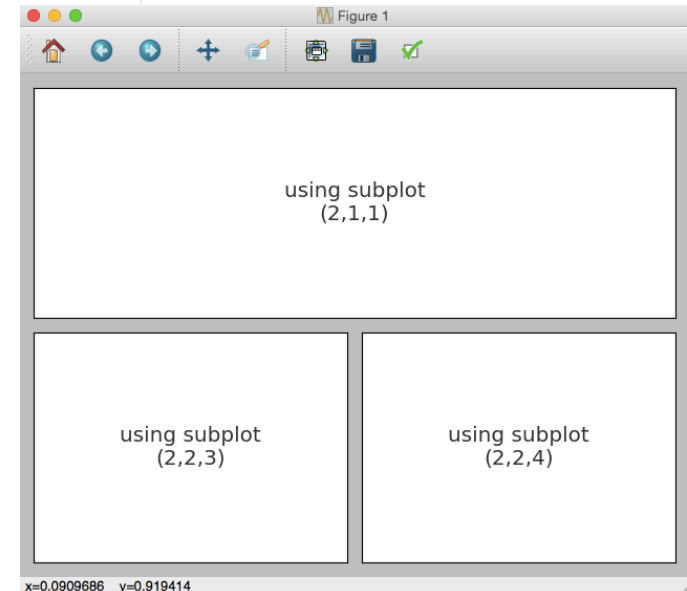
# Plotting tools

- Plotting tools:

*... so far we saw that:*

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

```
207 # subplot example:
208 plb.subplot(2, 1, 1)
209 plb.xticks(), plb.yticks()
210 plb.text(0.5, 0.5, 'using subplot\n(2,1,1)', ha='center', va='center',
211         size=18, alpha=.8)
212
213 plb.subplot(2, 2, 3)
214 plb.xticks(), plb.yticks()
215 plb.text(0.5, 0.5, 'using subplot\n(2,2,3)', ha='center', va='center',
216         size=18, alpha=.8)
217
218 plb.subplot(2, 2, 4)
219 plb.xticks(), plb.yticks()
220 plb.text(0.5, 0.5, 'using subplot\n(2,2,4)', ha='center', va='center',
221         size=18, alpha=.8)
222
223 plb.tight_layout() # makes the squares tighter to one another
224 plb.show()
```



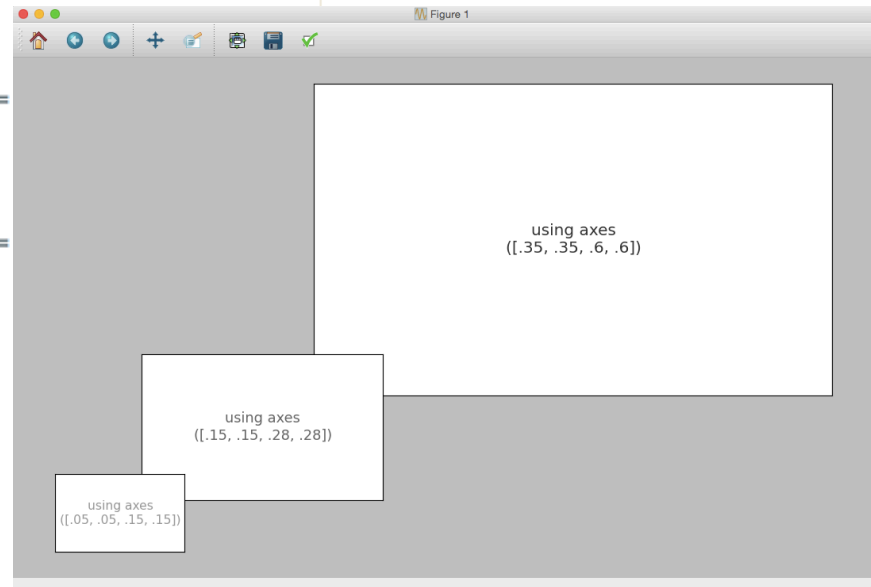
# Plotting tools

- Plotting tools:

*... so far we saw that:*

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

```
226 # axes example:
227 plb.axes([.35, .35, .6, .6])
228 plb.xticks(), plb.yticks()
229 plb.text(.5, .5, 'using axes\n([.35, .35, .6, .6])', ha='center', va='center',
230         size=18, alpha=.8)
231
232 plb.axes([.15, .15, .28, .28])
233 plb.xticks(), plb.yticks()
234 plb.text(.5, .5, 'using axes\n([.15, .15, .28, .28])', ha=
235         size=16, alpha=.6)
236
237 plb.axes([.05, .05, .15, .15])
238 plb.xticks(), plb.yticks()
239 plb.text(.5, .5, 'using axes\n([.05, .05, .15, .15])', ha=
240         size=14, alpha=.4)
241
242 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 current axes
- *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 implicitly is nice and quick, but offers limited usage
- explicit figure reference will provide more control over the display, while taking full advantage of figure, subplot, and axes

# Class exercise

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. Create a list A of 600 random numbers bound between (0:10)
4. Create an array B 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