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
- 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 *project discussion*

- **Matplotlib**
- What is Matplotlib?
- Basic plotting
- Tools: title, labels, legend, axis, points, subplots, etc.
- Advanced plotting: scatter, pie, bar, 3D plots, etc. *project examples,* HW2

NumPy arrays

- NumPy arrays
  - array attributes

T	Same as self.transpose(), except that self is returned if self.ndim < 2.
data	Python buffer object pointing to the start of the array's data.
dtype	Data-type of the array elements.
flags	Information about the memory layout of the array.
flatten	A 1-D iterator over the array.
imag	The imaginary part of the array.
real	The real part of the array.
size	Number of elements in the array.
itemsize	Length of one array element in bytes.
nbytes	Total bytes consumed by the elements of the array.
ndim	Number of array dimensions.
shape	Tuple of array dimensions.
strides	Tuple of bytes to step in each dimension when traversing an array.
ctypes	An object to simplify the interaction of the array with the ctypes module.
base	Base object if memory is from some other object.

*source – NumPy reference

NumPy arrays

- NumPy arrays

  Examples:

```
In [43]: f = np.ndarray(shape=(2,3,2), dtype=complex)

In [44]: f
Out[44]:
array([[[  0.00000000e+000 -2.00000013e+000j,
           2.12215769e-314 +9.88131292e-324j],
        [  0.00000000e+000 +0.00000000e+000j,
           0.00000000e+000 -9.84629069e+109j],
        [  0.00000000e+000 +0.00000000e+000j,
           2.25697366e-314 +0.00000000e+000j]],

       [[  0.00000000e+000 +2.25697468e-314j,
           0.00000000e+000 +0.00000000e+000j],
        [ -2.58861351e-056 +0.00000000e+000j,
           0.00000000e+000 -2.05241193e-191j],
        [  2.12381808e-314 +2.25685768e-314j,
          -4.57473710e+035 +2.24500133e-314j]]])

In [45]: f.real
Out[45]:
array([[[  0.00000000e+000,   2.12215769e-314],
        [  0.00000000e+000,   0.00000000e+000],
        [  0.00000000e+000,   2.25697366e-314]],

       [[  0.00000000e+000,   0.00000000e+000],
        [ -2.58861351e-056,   0.00000000e+000],
        [  2.12381808e-314,  -4.57473710e+035]]])

In [46]: f.real.T
Out[46]:
array([[[  0.00000000e+000,   0.00000000e+000],
        [  0.00000000e+000,  -2.58861351e-056],
        [  0.00000000e+000,   2.12381808e-314]],

       [[  2.12215769e-314,   0.00000000e+000],
        [  0.00000000e+000,   0.00000000e+000],
        [  2.25697366e-314,  -4.57473710e+035]]])
```

… try it in class

NumPy arrays

- NumPy arrays

Examples:

```
In [47]: f.imag.flags
Out[47]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False

In [48]: f.imag.data
Out[48]: <memory at 0x110298ce0>

In [49]: f.real.dtype
Out[49]: dtype('float64')

In [50]: f.dtype
Out[50]: dtype('complex128')

In [51]: f.shape
Out[51]: (2, 3, 2)

In [52]: f.T.shape
Out[52]: (2, 3, 2)

In [53]: f.size
Out[53]: 12

In [54]: f.itemsize
Out[54]: 16

In [55]: f.nbytes
Out[55]: 192

In [56]: f.ndim
Out[56]: 3
```

Note - it can be seen that the attributes of ndarray can be used in a nested fashion

… try it in class

NumPy arrays

- NumPy arrays
  - flags – gives information about the memory layout of the array

C_CONTIGUOUS (C)	The data is in a single, C-style contiguous segment.
F_CONTIGUOUS (F)	The data is in a single, Fortran-style contiguous segment.
OWNDATA (O)	The array owns the memory it uses or borrows it from another object.
WRITEABLE (W)	The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.
ALIGNED (A)	The data and all elements are aligned appropriately for the hardware.
UPDATEIFCOPY (U)	This array is a copy of some other array. When this array is de-allocated, the base array will be updated with the contents of this array.
FNC	F_CONTIGUOUS and not C_CONTIGUOUS.
FORC	F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).
BEHAVED (B)	ALIGNED and WRITEABLE.
CARRAY (CA)	BEHAVED and C_CONTIGUOUS.
FARRAY (FA)	BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

*source – NumPy reference

NumPy arrays

- NumPy arrays
  - flatten – returns a copy of the same flattened array in one dimension

```
In [57]: g = np.arange(12, 24).reshape(3, 4)

In [58]: g
Out[58]:
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])

In [59]: g[:,:]
Out[59]:
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])

In [60]: g.flat[6]
Out[60]: 18

In [61]: g.flat[9]
Out[61]: 21

In [62]: g.flat[:]
Out[62]: array([12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23])

In [63]: g.flatten()
Out[63]: array([12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23])

In [64]: g.T.flat[6]
Out[64]: 14
```

```
In [33]: g.flatten(order='C')
Out[33]: array([12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23])

In [34]: g.flatten(order='F')
Out[34]: array([12, 16, 20, 13, 17, 21, 14, 18, 22, 15, 19, 23])
```

```
In [18]: g.T
Out[18]:
array([[12, 16, 20],
       [13, 17, 21],
       [14, 18, 22],
       [15, 19, 23]])
```

... try it in class

NumPy arrays

- NumPy arrays
  - shape – besides checking or specifying the shape of an array, by using the shape command we can also re-shape an array so long that we do not change the number of elements in it

Example:

```
In [65]: h = np.array([[12,34,41],[54,67,89],[102,13,45],[78,456,218]])

In [66]: h
Out[66]:
array([[ 12,  34,  41],
       [ 54,  67,  89],
       [102,  13,  45],
       [ 78, 456, 218]])

In [67]: h.shape
Out[67]: (4, 3)

In [68]: h.shape = (2,6)

In [69]: h
Out[69]:
array([[ 12,  34,  41,  54,  67,  89],
       [102,  13,  45,  78, 456, 218]])

In [70]: h.shape = (3,6)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-70-76a181f81944> in <module>()
----> 1 h.shape = (3,6)

ValueError: total size of new array must be unchanged
```

NumPy arrays

- NumPy arrays
  - strides – represents the number of bytes (8-bit each) needed to travel in each direction (in memory) in a multidimensional array in order to get to certain element in that array along a given axis

Example:

Note – the given array *i* is stored in a continuous block of memory of:

480 bytes, or:

```
In [154]: (3*4*5)*8
Out[154]: 480
```

Python

```
In [71]: i = np.reshape(np.arange(3*4*5), (5,3,4))

In [72]: i
Out[72]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]],

       [[24, 25, 26, 27],
        [28, 29, 30, 31],
        [32, 33, 34, 35]],

       [[36, 37, 38, 39],
        [40, 41, 42, 43],
        [44, 45, 46, 47]],

       [[48, 49, 50, 51],
        [52, 53, 54, 55],
        [56, 57, 58, 59]]])

In [73]: np.shape(i)
Out[73]: (5, 3, 4)
```

same as:

```
In [111]: i.shape
Out[111]: (5, 3, 4)
```

```
In [139]: i.nbytes
Out[139]: 480

In [140]: i[0]
Out[140]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [141]: i[0].nbytes
Out[141]: 96

In [142]: i[0][0]
Out[142]: array([0, 1, 2, 3])

In [143]: i[0][0].nbytes
Out[143]: 32

In [144]: i[0][0][0]
Out[144]: 0

In [145]: i[0][0][0].nbytes
Out[145]: 8
```

NumPy arrays

- NumPy arrays
  - strides

Example:

```
In [74]: i[4][2][1]
Out[74]: 57

In [75]: i[4,2,1]
Out[75]: 57

In [76]: np.dtype(i[4,2,1])
Out[76]: dtype('int64')

In [77]: i
Out[77]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]],

       [[24, 25, 26, 27],
        [28, 29, 30, 31],
        [32, 33, 34, 35]],

       [[36, 37, 38, 39],
        [40, 41, 42, 43],
        [44, 45, 46, 47]],

       [[48, 49, 50, 51],
        [52, 53, 54, 55],
        [56, 57, 58, 59]]])

In [78]: i.strides
Out[78]: (96, 32, 8)
```

Note – you can easily refer to an element from the array, knowing its position as shown in lines Out [74],[75]

... try it in class

NumPy arrays

- NumPy arrays
  - strides

Example:

```
In [79]: np.array([4,2,1])
Out[79]: array([4, 2, 1])

In [80]: np.array([4,2,1]) * i.strides
Out[80]: array([384,  64,   8])

In [81]: sum(np.array([4,2,1]) * i.strides)
Out[81]: 456

In [82]: i.itemsize
Out[82]: 8

In [83]: sum(np.array([4,2,1]) * i.strides)/i.itemsize
Out[83]: 57.0

In [84]: i
Out[84]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]],

       [[24, 25, 26, 27],
        [28, 29, 30, 31],
        [32, 33, 34, 35]],

       [[36, 37, 38, 39],
        [40, 41, 42, 43],
        [44, 45, 46, 47]],

       [[48, 49, 50, 51],
        [52, 53, 54, 55],
        [56, 57, 58, 59]]])
```

```
In [166]: np.array([4,2,1])*i.strides/i.itemsize
Out[166]: array([48.,  8.,  1.])
```

Note – … you can calculate it in an iterative way shown in line Out[83]

… try it in class

NumPy arrays

- NumPy arrays
  - transpose – transpose can easily be performed by used a specific attribute (command)

Example:

note:
.T and .transpose()
do the same job!

… try it in class

```
In [85]: i.transpose
Out[85]: <function ndarray.transpose>

In [86]: i.transpose()
Out[86]:
array([[[ 0, 12, 24, 36, 48],
        [ 4, 16, 28, 40, 52],
        [ 8, 20, 32, 44, 56]],

       [[ 1, 13, 25, 37, 49],
        [ 5, 17, 29, 41, 53],
        [ 9, 21, 33, 45, 57]],

       [[ 2, 14, 26, 38, 50],
        [ 6, 18, 30, 42, 54],
        [10, 22, 34, 46, 58]],

       [[ 3, 15, 27, 39, 51],
        [ 7, 19, 31, 43, 55],
        [11, 23, 35, 47, 59]]])

In [87]: np.shape(i.transpose())
Out[87]: (4, 3, 5)
```

NumPy with other languages

- NumPy arrays
  - ctypes:
    - this module is part of the standard Python distribution package
    - it is used for shared C-libraries, in case you have some useful code written in C and would like to put a Python wrapper around it to incorporate a specific routine written in C in your code
    - this possibility opens up a great number of already well written and tested C routines
    - the problem when using this module however is that it can lead to nasty crashes because of poor type checking

    Example:

    a problem can occur when you pass an array as a pointer to a raw memory location and you forget to check if the subroutine may access memory outside of the array boundaries

NumPy with other languages

- NumPy arrays
  - ctypes:
    - when using *ctypes* remember that this approach uses a raw memory location to a compiled code and it may not be error prone to user mistakes
    - good knowledge of the shared library and this module is a must
    - this approach most times requires extra Python code to handle errors of different kind to:
      - check for the data types
      - array boundaries of the passes objects
    - this however will slow down the interface because of all additional checking and type conversion (C to Python) that is necessary
    - this tool is for people with strong Python skills, but weak C knowledge

NumPy with other languages

- NumPy arrays
  - ctypes:
    - to use *ctypes* you must have the following:
      - have a library to be shared
      - load the library to be shared
      - convert the Python objects to *ctypes* arguments that can be interpreted correctly
      - call the function from the library containing the *ctypes* arguments
    - when using *ctypes* some of the basic attributes that can be used are:
      - data, shape and strides (... for more attributes please refer to the NumPy guide)
    - one should be careful when using temporary arrays or arrays constructed on the fly, because they return a pointer to an invalid memory location since it has been de-allocated as soon as the next Python statement is reached

Examples:

a) (a+b).ctypes – wrong , because the array created as (a+b) is de-allocated before the next statement

b) c = (a+b).ctypes – correct, because *c* will have a reference to the array

NumPy with other languages

- NumPy arrays
  - ctypes:

Examples:

```
In [88]: import numpy as np

In [89]: j = np.array([[12, 34, 99, 32],[41, 52, 45, 16],[64, 88, 67, 58]])

In [90]: j
Out[90]:
array([[12, 34, 99, 32],
       [41, 52, 45, 16],
       [64, 88, 67, 58]])

In [91]: j.ctypes
Out[91]: <numpy.core._internal._ctypes at 0x105999588>

In [92]: j.ctypes.data
Out[92]: 4466631472

In [18]: j.ctypes.shape._length_
Out[18]: 2

In [19]: j.ctypes.strides._length_
Out[19]: 2

In [20]: j.ctypes.shape._type_
Out[20]: ctypes.c_long
```

... try it in class

NumPy with other languages

- NumPy arrays

  – ctypes: Example:

1. begin with writing your C library and save the file ‘ctypes_lib.c’:

```c
#include <stdio.h>

void myprint(void);
void myprint()
{
    printf("This is ctypes example in Python\n");
}
```

2. install your gcc if you don’t have one (skip this step if you do):

   - PC: find a compiler and install using the .exe file. Try using Cygwin - a Unix-like environment on Win
   - Mac OS X in the terminal type: xcode-select –install

3. you need to compile the file as shared library using this notation:

   - PC: $ gcc -shared -Wl,-soname, ctypes_lib -o ctypes_lib.so -fPIC ctypes_lib.c
   - Mac OS X: $ gcc -shared -Wl,-install_name, ctypes_lib.so -o ctypes_lib.so -fPIC ctypes_lib.c

```
Macintosh:lecture4 alex$ gcc -shared -Wl,-install_name,ctypes_lib.so -o ctypes_lib.so -fPIC ctypes_lib.c
[Macintosh:lecture4 alex$
```

NumPy with other languages

- NumPy arrays
  - ctypes: Example:

4. Create your ctypes Python wrapper module ‘ctypes_lib_tester.py’ and execute it:

```python
## Ctypes example of using a C file code:

import ctypes

c_test_lib = ctypes.CDLL('ctypes_lib.so')
c_test_lib.myprint()
```

5. The result should be:

```
Python
This is ctypes example in Python
```

6. If you run:

```
In [1]: c_test_lib.myprint()
Out[1]: 33
```

this only prints the number of characters in the ‘c’ library, so for the text:
“This is ctypes example in Python\n” there are 33, including the end of line character ‘\n’

NumPy with other languages

- NumPy arrays

  – ctypes: Example:

7. If you compile and execute the library in the terminal here is the result:

```
lecture4 — -bash — 114×16
Macintosh:lecture4 alex$ gcc -shared -Wl,-install_name,ctypes_lib.so -o ctypes_lib.so -fPIC ctypes_lib.c
Macintosh:lecture4 alex$ ls -la
total 88
drwxr-xr-x  10 alex  501   340 Oct 24 13:55 .
drwxr-xr-x   9 alex  501   306 Nov 19  2015 ..
-rw-r--r--@  1 alex  501  6148 Oct 24 13:45 .DS_Store
-rw-r--r--   1 alex  501   613 Oct 22 13:52 class excercise.py
-rw-r--r--   1 alex  501   110 Oct 24 13:53 ctypes_lib.c
-rwxr-xr-x   1 alex  501  8376 Oct 24 13:55 ctypes_lib.so
-rw-r--r--   1 alex  501   122 Oct 24 13:55 ctypes_lib_tester.py
-rw-r--r--   1 alex  501  1361 Oct 24 12:55 lecture4 - inverting a non-square matrix using SVD - pseudoinverse.py
-rw-r--r--   1 alex  501    81 Oct 22 13:55 my_array
-rw-r--r--   1 alex  501   439 Oct 19 20:49 speed_test_list_ndarray.py
Macintosh:lecture4 alex$ python ctypes_lib_tester.py
This is ctypes example in Python
Macintosh:lecture4 alex$
```

NumPy with other languages

C/C++

There a various tools which make it easier to bridge the gap between Python and C/C++:

» Pyrex - write your extension module on Python

» Cython -- Cython -- an improved version of Pyrex

» CXX - PyCXX - helper lib for writing Python extensions in C++

» ctypes is a Python module allowing to create and manipulate C data types in Python. These can then be passed to C-functions loaded from dynamic link libraries.

» elmer - compile and run python code from C, as if it was written in C

» PicklingTools is a collection of libraries for exchanging Python Dictionaries between C++ and Python.

» weave - include C code lines in Python program (deprecated in favor of Cython)

» ackward exposes parts of Python's standard library as idiomatic C++

» CFFI - interact with almost any C code from Python, based on C-like declarations that you can often copy-paste from header files or documentation.

Source: https://wiki.python.org/moin/IntegratingPythonWithOtherLanguages#Java

NumPy with other languages

Java

- Jython - Python implemented in Java
- JPype - Allows Python to run java commands
- Jepp - Java embedded Python
- JCC - a C++ code generator for calling Java from C++/Python
- Javabridge - a package for running and interacting with the JVM from CPython
- py4j - Allows Python to run java commands.
- voc - Part of BeeWare suite. Converts python code to Java bytecode.
- p2j - Converts Python code to Java. No longer developed.

Source: https://wiki.python.org/moin/IntegratingPythonWithOtherLanguages#Java

NumPy with other languages

Perl

See http://www.faqts.com/knowledge_base/view.phtml/aid/17202/fid/1102

- PyPerl http://search.cpan.org/dist/pyperl/
- Inline::Python
- PyPerlish - Perl idioms in Python

For converting/porting Perl code to Python the tool 'Bridgekeeper' http://www.crazy-compilers.com/bridgekeeper/ may be handy.

PHP

- PiP (Python in PHP) http://www.csh.rit.edu/~jon/projects/pip/
- PHP "Serialize" in Python http://hurring.com/scott/code/python/serialize/ (broken link; see the Web Archive Wayback Machine for the latest working version)

R

- RPy http://rpy.sourceforge.net
- RSPython http://www.omegahat.net/RSPython

Source: https://wiki.python.org/moin/IntegratingPythonWithOtherLanguages#Java

NumPy arrays

- NumPy arrays
  - base:
    - is an attribute used when we need to keep track of the memory reference of the original object owner in case two objects are referring to the same memory location
    - it is a way of NumPy to keep track of the data source in memory for any given array

Example:

```
In [95]: k = np.array([[12, 34, 99],[41, 52, 45],[64, 88, 67]])

In [96]: k
Out[96]:
array([[12, 34, 99],
       [41, 52, 45],
       [64, 88, 67]])

In [97]: k.base is None
Out[97]: True

In [98]: l = k[0:2]

In [99]: l
Out[99]:
array([[12, 34, 99],
       [41, 52, 45]])

In [100]: l.base is k
Out[100]: True
```

... try it

Indexing and slicing of arrays

- Indexing and slicing of arrays (recap)
  - indexing:
    - so far we saw indexing of arrays in many occasions
    - indexing in NumPy arrays begins from ‘0’ and is runs in an scalar/integer progression
  - slicing:
    - a slice is an object referring to a portion (slice) of an array
    - the newly created slice is an array generated by slicing the original array and it always represents a view of the latter

Indexing and slicing of arrays

- Indexing and slicing of arrays

Examples:

```
In [101]: m = np.array([[12, 34, 99],[41, 52, 45],[64, 88, 67]])

In [102]: m
Out[102]:
array([[12, 34, 99],
       [41, 52, 45],
       [64, 88, 67]])

In [103]: m.shape
Out[103]: (3, 3)

In [104]: m[0:]       # slicing
Out[104]:
array([[12, 34, 99],
       [41, 52, 45],
       [64, 88, 67]])

In [105]: m[1]        # indexing/slicing
Out[105]: array([41, 52, 45])

In [106]: m[1:2]      # slicing
Out[106]: array([[41, 52, 45]])

In [107]: m[1][0]     # indexing
Out[107]: 41

In [108]: m[:3]       # slicing
Out[108]:
array([[12, 34, 99],
       [41, 52, 45],
       [64, 88, 67]])

In [109]: m[2,1]      # indexing
Out[109]: 88

In [110]: m[-1,-3]    # slicing - representing element (-1+3=2, -3+3=0)
Out[110]: 64
```

… try it in class

Numpy - matrix, array and ndarray

- 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 - matrix, array and ndarray

- 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)
```

Discussion

- Discussion

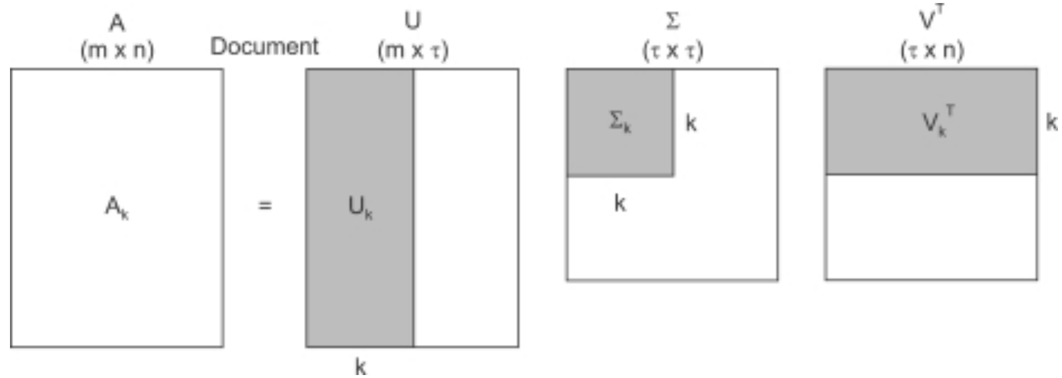
  matrix inversion:

  1. by definition a matrix is commutative with its inverse on multiplication:
  $A_{[mxn]} * A_{[nxm]}^{-1} = A_{[nxm]}^{-1} * A_{[mxn]} = I$

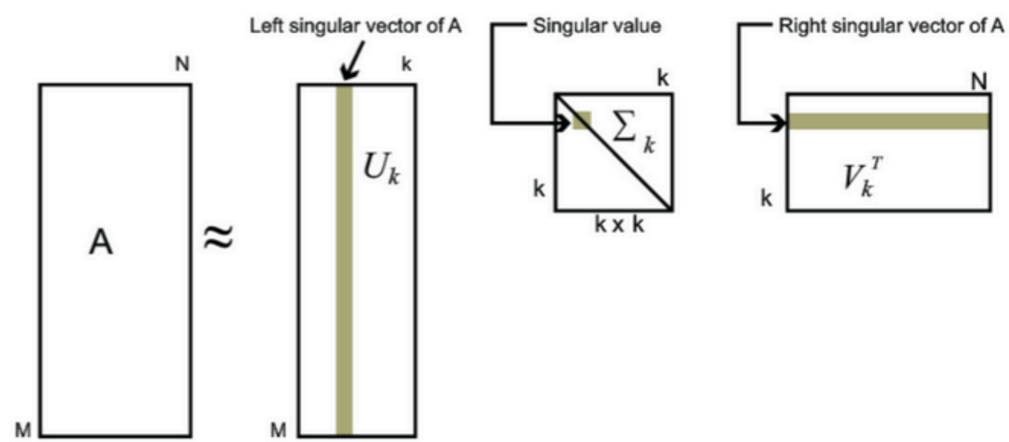
     so, it must be that m=n!

  2. A non-square matrix inverse is possible using SVD:
     There exists a left inverse U and a right inverse V that is defined for **all** matrices including non-square matrices

$$A = U \Sigma V^T$$



$$A \approx U_k \Sigma_k V_k^T$$



Discussion

- Discussion

matrix inversion:

1. by definition a matrix is commutative with its inverse on multiplication:

$$A_{[mxn]}*A_{[nxm]}^{-1}=A_{[nxm]}^{-1}*A_{[mxn]}=I$$

so, it must be that m=n!

2. A non-square matrix inverse is possible using SVD:

There exists a left inverse U and a right inverse V that is defined for **all** matrices including non-square matrices

```
# Using Singular Value Decomposition (SVD) for manually performing a pseudoinverse on a non-square matrix:
# It is not the actual inverse matrix, but the "best approximation" of such in the sense of least squares

from numpy import random, matrix, linalg, diag, allclose, dot

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

# 1-3. Using the SVD function will return:
# U - a matrix with columns = the eigenvectors (L) of the A*A.T
#     holds Left-singular vectors
# s - a diagonal matrix with diagonal = the singular values of matrix A:
#     the singular (diagonal) values in s are square roots of eigenvalues from U and V
#     Σ+ is the pseudoinverse of Σ, which is formed by replacing every non-zero
#     diagonal entry by its reciprocal and transposing the resulting matrix
# V - a matrix with columns = the eigenvectors (R) of the A.T*A
#     holds Right-singular vectors
# U and V - must preserve the properties of the original matrix A, so they are orthogonal
U,s,V = linalg.svd(A, full_matrices=False)

# Constrcut a giagonal matrix 'S', from the giagonal 's':
S = diag(s)

# 2-3. Invert the square diagonal matrix by inverting each diagonal element:
S[0,0], S[1,1], S[2,2] = 1/diag(S[0:3,0:3])

# 3-3. Now we use the SVD elements to obtain the pseudo-inverse of matrix A:
X = dot(U, dot(S, V))
X = X.T # Final step: we must transpose

# Check each matrix:
A.shape, U.shape, S.shape, V.shape

# Comparison test 1:
A.I-X

# Comparison test 2:
allclose(A.I, X)
```

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:

```
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:

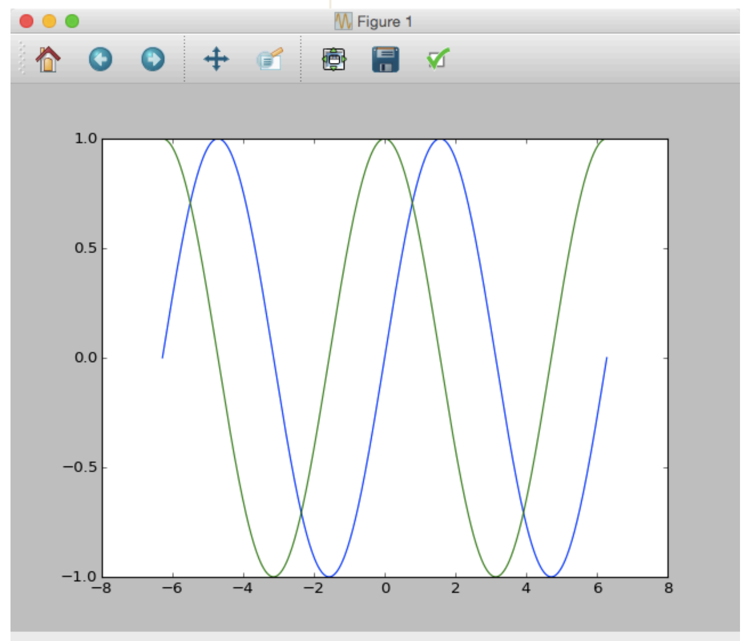
```
In [4]: a = plb.np.array([[12,34],[41,54]])

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()
```

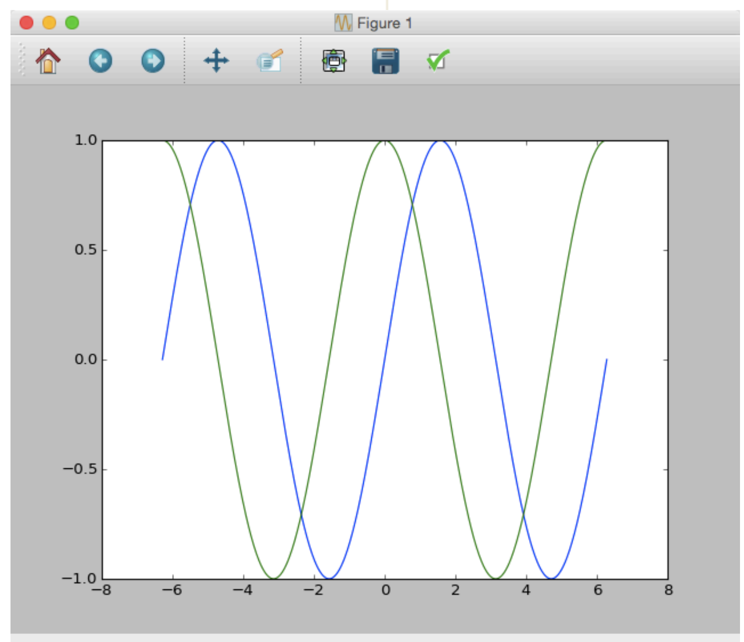


Basic plotting

- Basic plotting: *comparison*

```python
## 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):

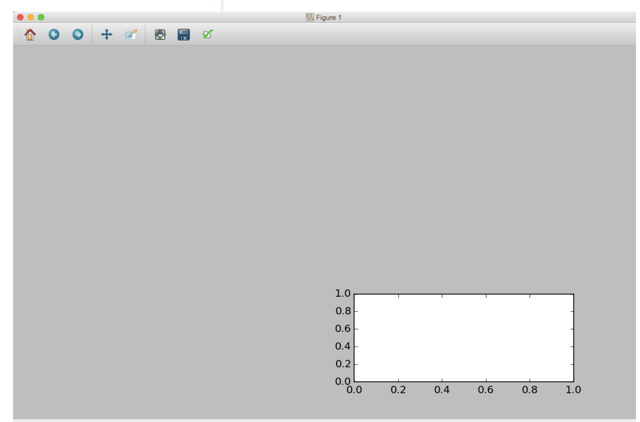
  plb.figure(figsize=(10, 6), dpi=120) # this line will create an empty window

  – 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:

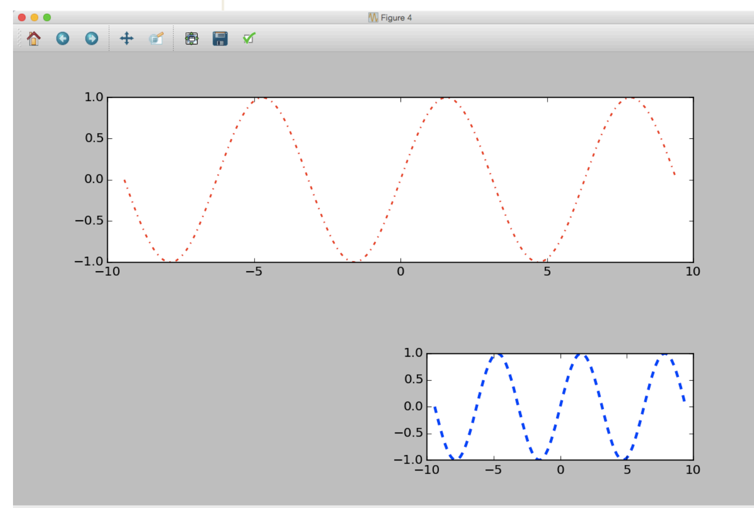
```
In [7]: import pylab as plb

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*

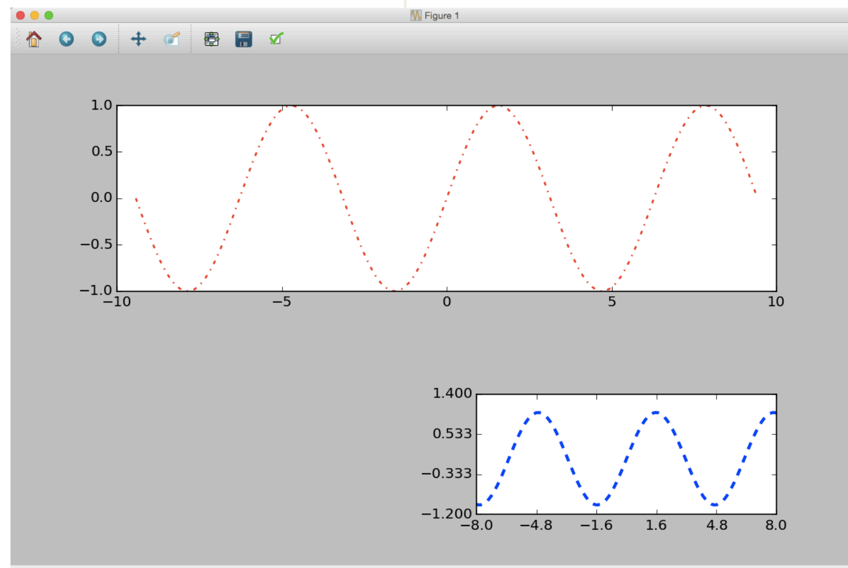
```python
## Basic plotting tools:
import pylab as plb
plb.figure(figsize=(10, 6), dpi=120)

# 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_cos, color="blue", linewidth=2.5, linestyle="--")
```



Plotting tools

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

```
# 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)
```

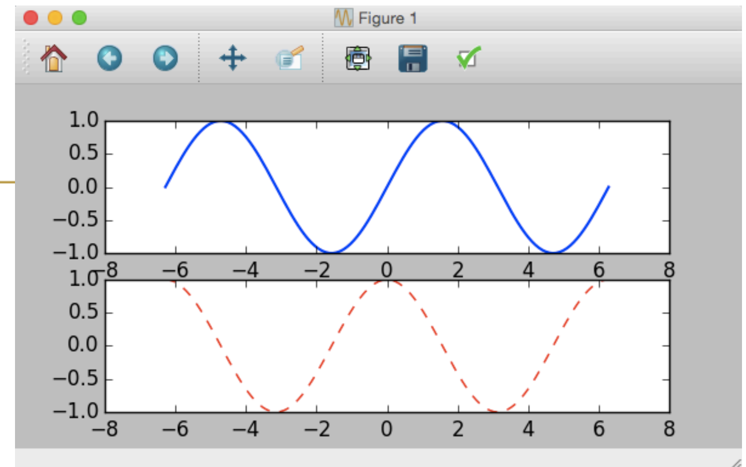


Plotting tools

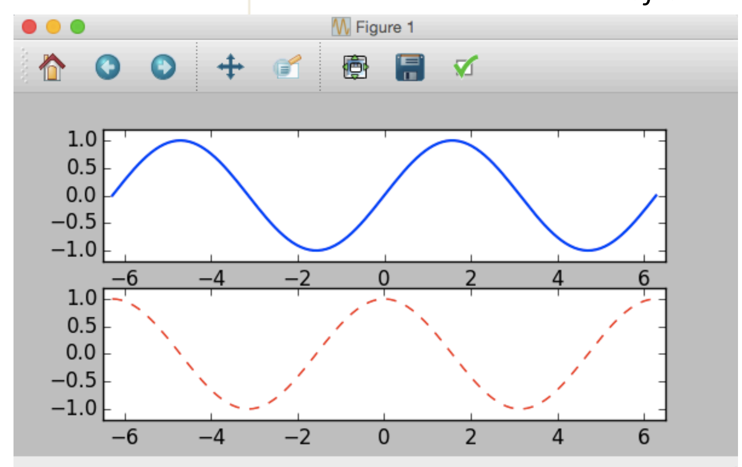
- Plotting tools: *changing plot limits*

```
## 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



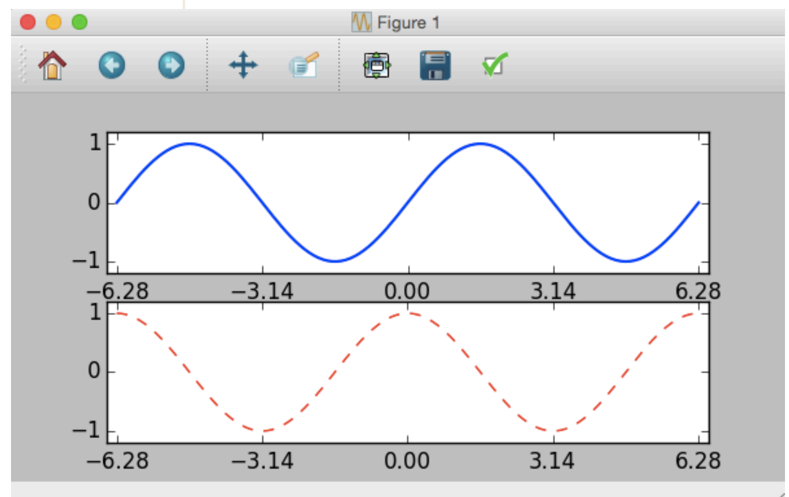
after



Plotting tools

- Plotting tools: *editing ticks*

```
## 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*

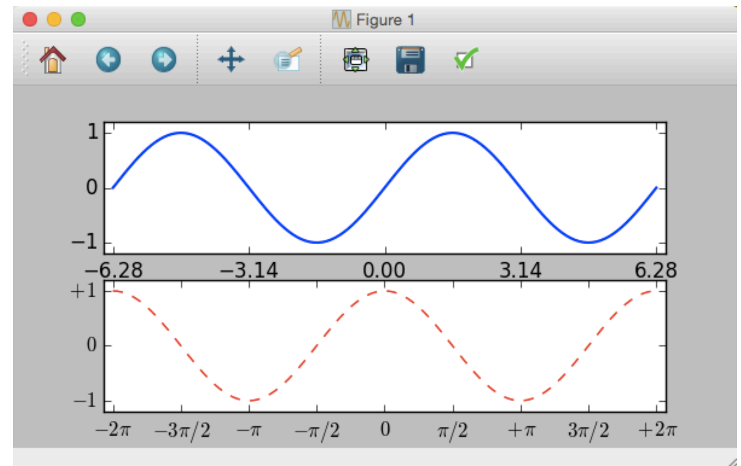
Now that we set the ticks correctly, we need to be a bit more explicit about what they represent, so we add the following code:

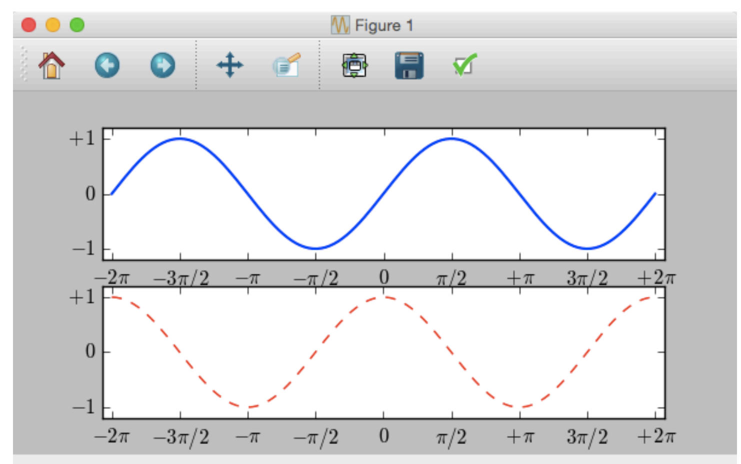
```
# adding x,y tick labels for plot (2,1,2):
plb.xticks([-plb.pi*2, -3*plb.pi/2, -plb.pi, -plb.pi/2, 0,
                plb.pi/2, plb.pi, 3*plb.pi/2, plb.pi*2],
          ['$-2\pi$', '$-3\pi/2$', '$-\pi$', '$-\pi/2$', '$0$', \
                '$\pi/2$', '$+\pi$', '$3\pi/2$', '$+2\pi$'])
plb.yticks([-1,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

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

line split

```
# 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],
          ['$-1$', '$0$', '$+1$'])
```

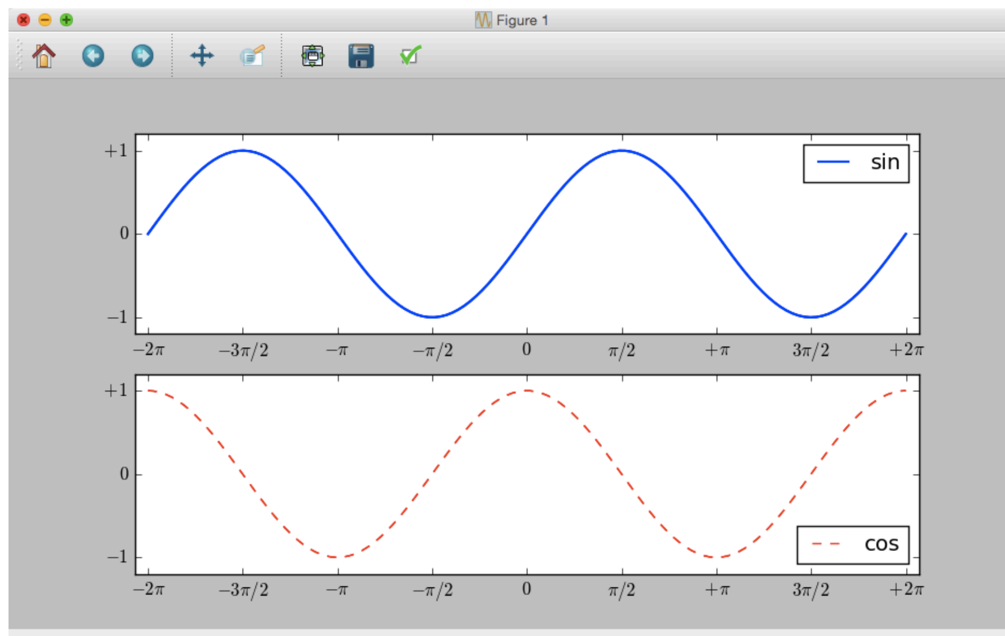




Plotting tools

- Plotting tools: *adding a legend*

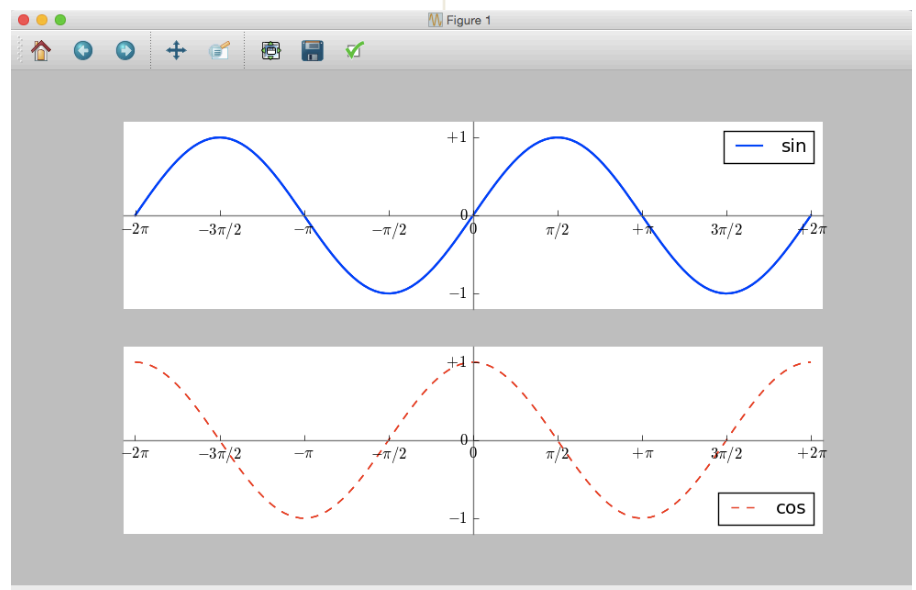
```
# adding a legend clarifying the plots:
plb.subplot(2,1,1)
plb.plot(d, d_sin, color="blue",  linewidth=1.5, linestyle="-", label="sin")
plb.legend(loc='upper right')
plb.subplot(2,1,2)
plb.plot(d, d_cos, color="red", linewidth=1, linestyle="--", label="cos")
plb.legend(loc='lower right')
```



Plotting tools

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

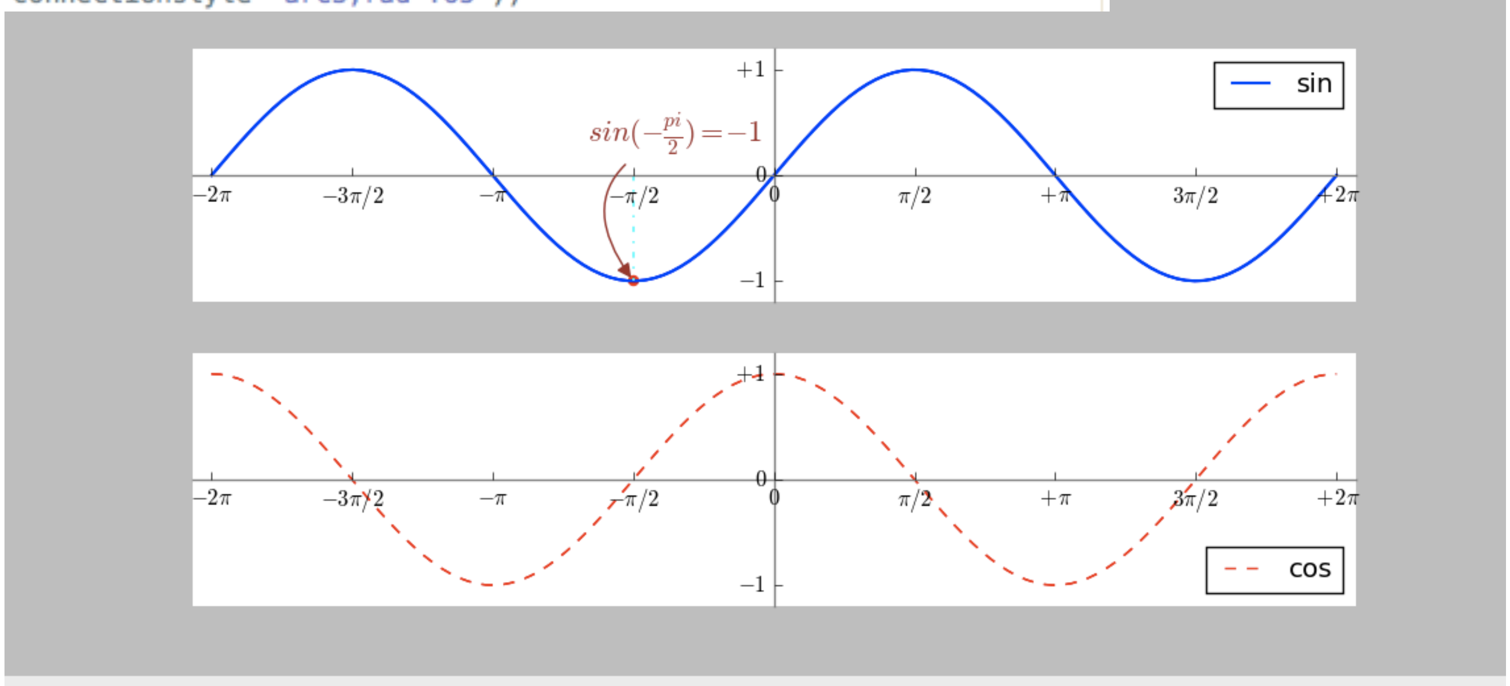
```
# setting the axis:
ax1 = plb.gca()  # gca - 'get current axis'
ax1.spines['top'].set_color('none')     # to get rid of the black border line
ax1.spines['bottom'].set_color('none') # to get rid of the black border line
ax1.spines['left'].set_color('none')   # to get rid of the black border line
ax1.spines['right'].set_color('none')  # 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*

```
# 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"))
```

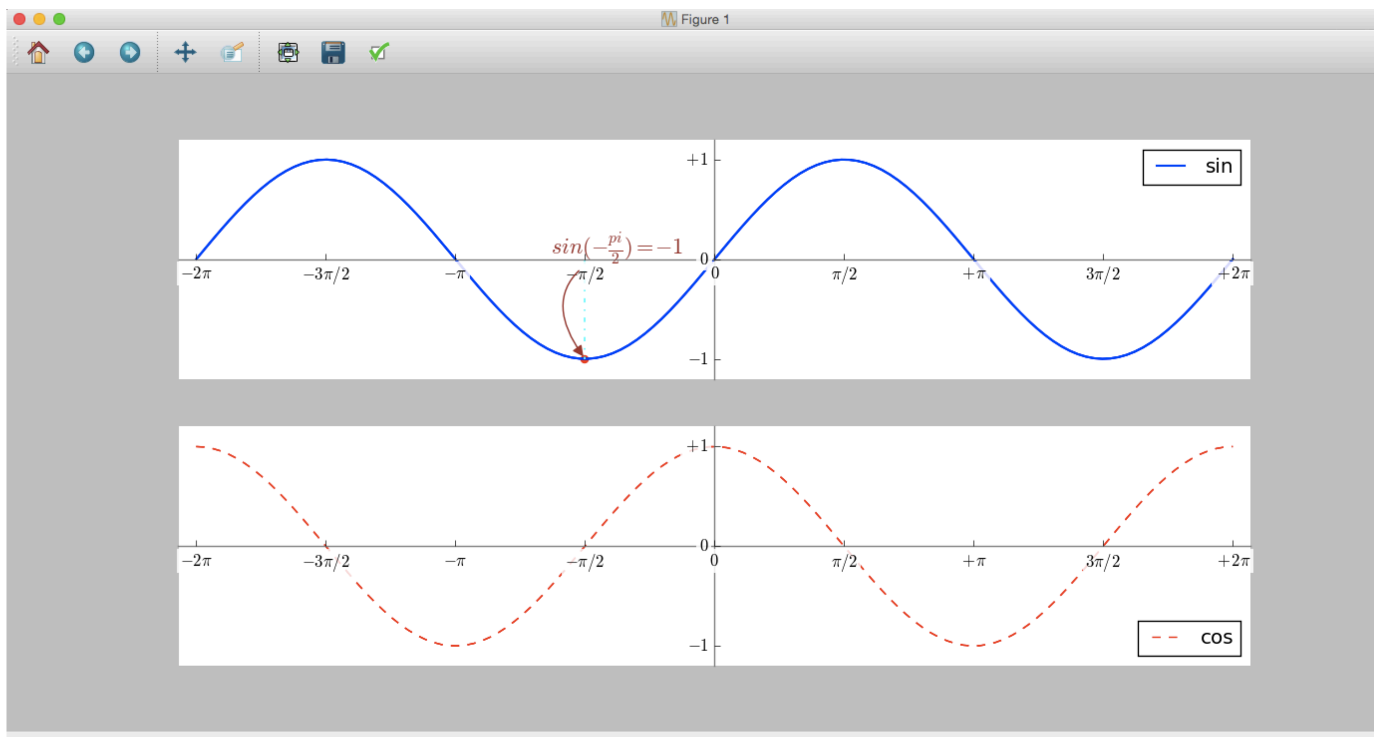


Plotting tools

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

```
# 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))
```

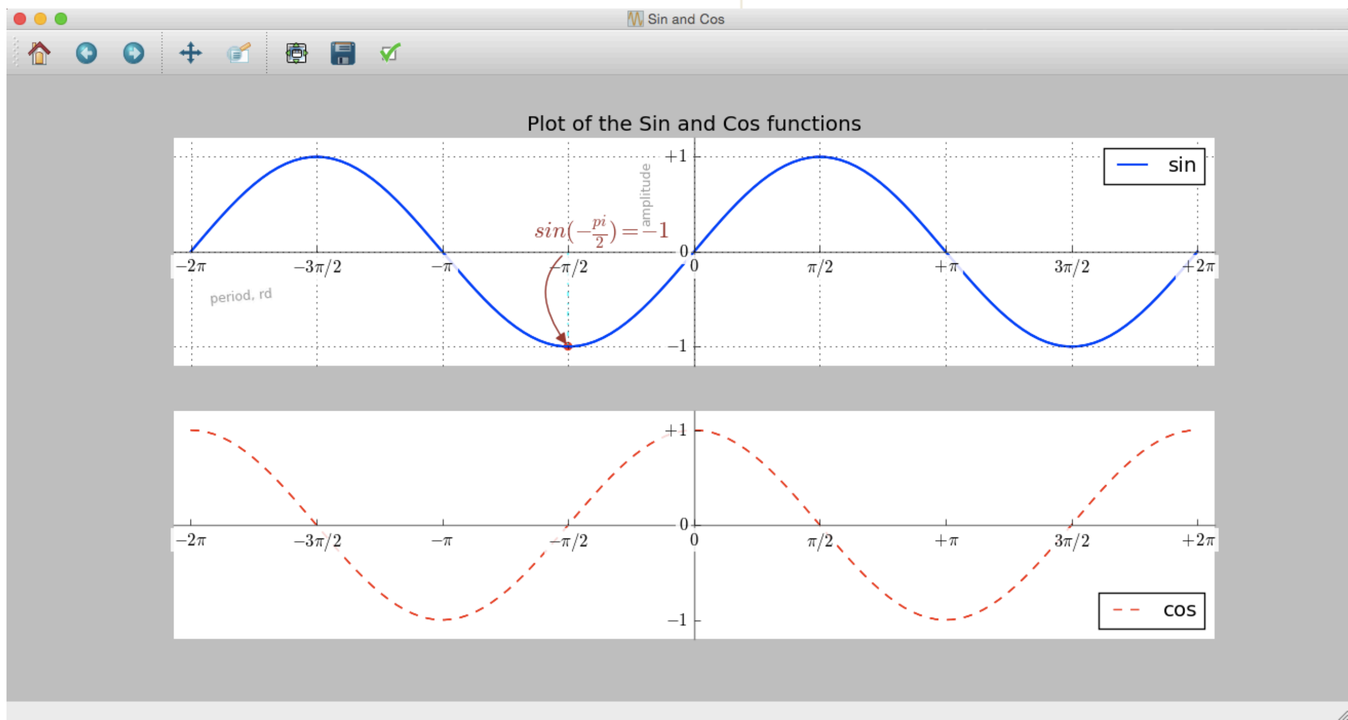
our tick labels are obscured by the plot lines running over them, so we need to make them more clear and visible



Plotting tools

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

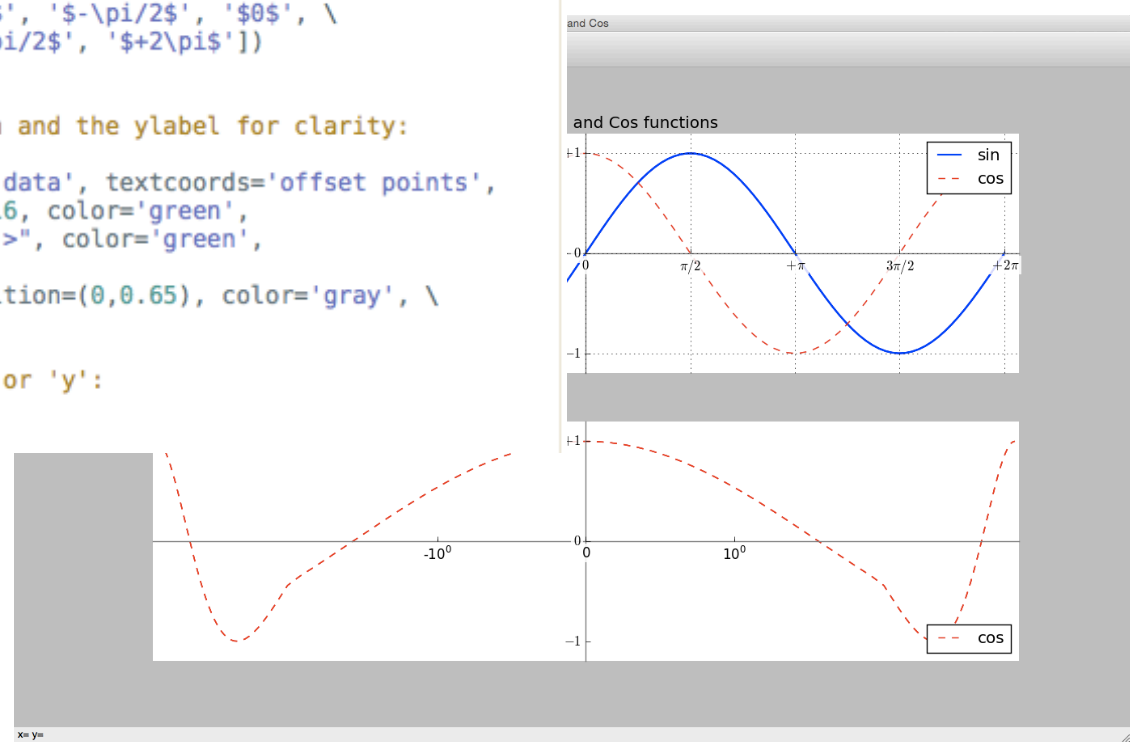
```
## adding some goodies:

# 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*

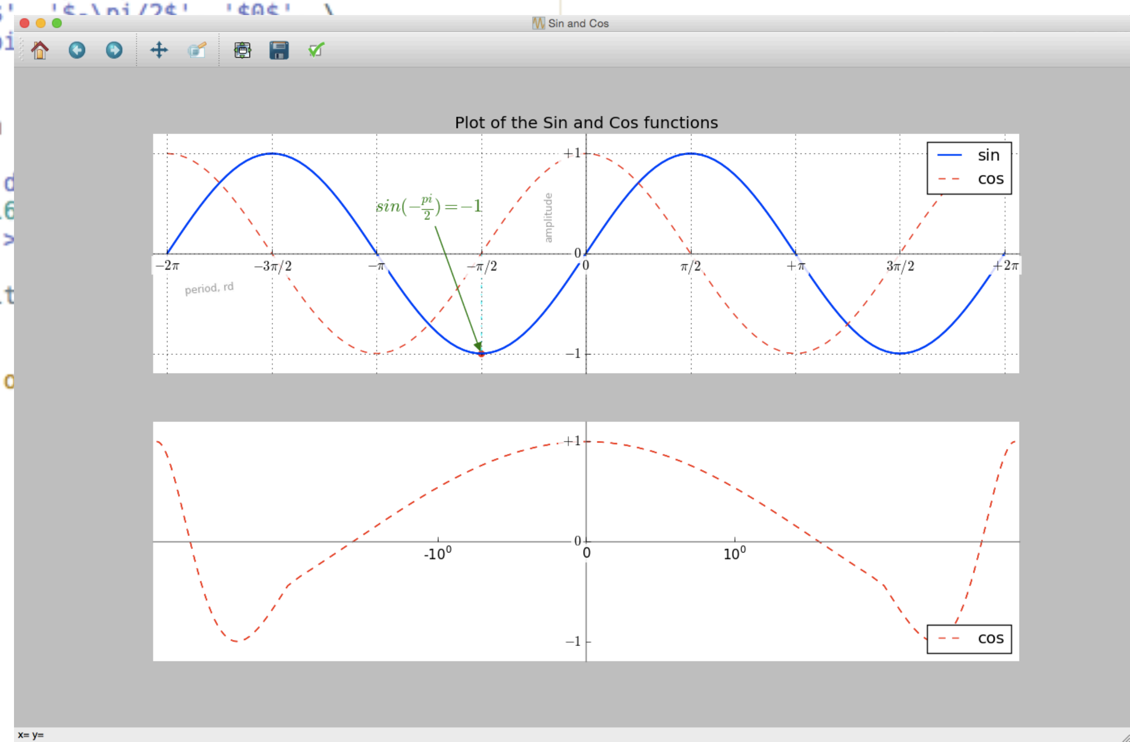
```
# hold so that another plot can be drawn on top of the current:
plb.hold(True)

# 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*

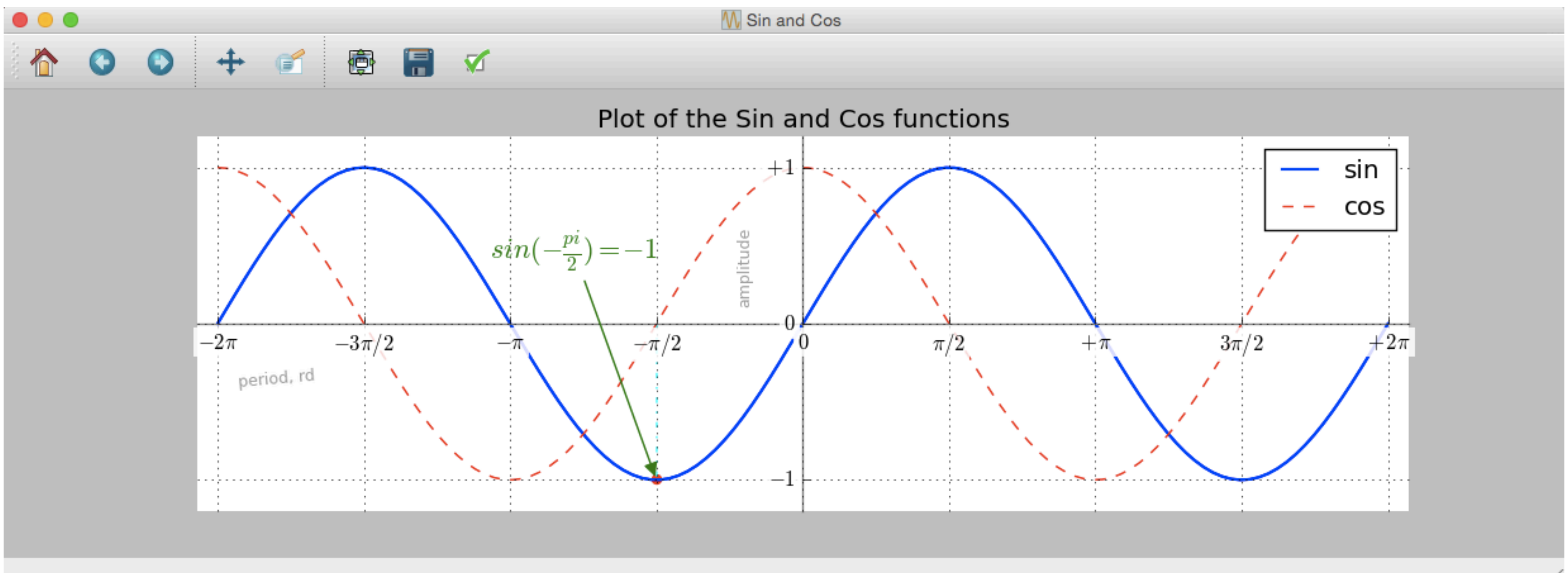
```
# hold so that another plot can be drawn on top of the current:
plb.hold(True)

# 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],
          ['$-2\pi$', '$-3\pi/2$', '$-\pi$', '$-\pi/2$', '$0$', \
                '$\pi/2$', '$+\pi$', '$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*

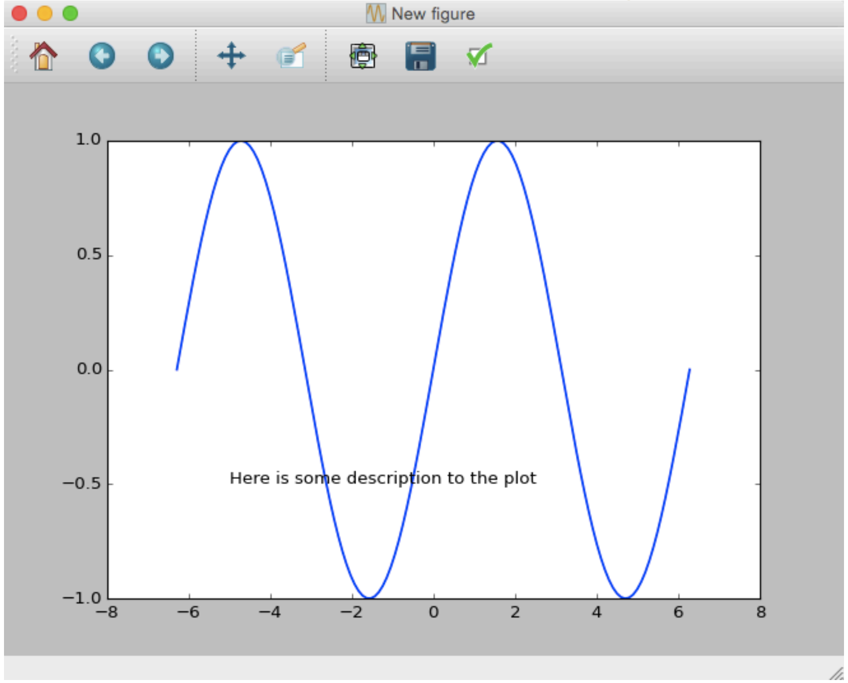
```
# to remove the second subplot at position (2,1,2) do this:
ax1.set_visible(False)
ax2.change_geometry(1,1,1)

# 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

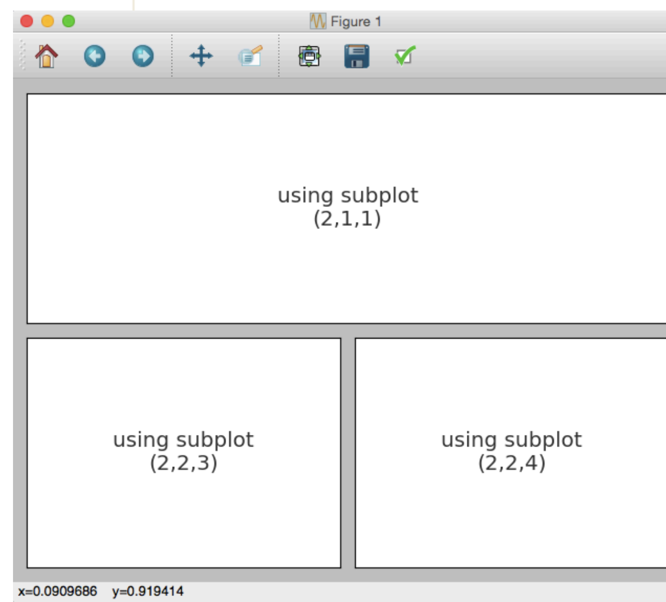
Plotting tools

- 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()
```



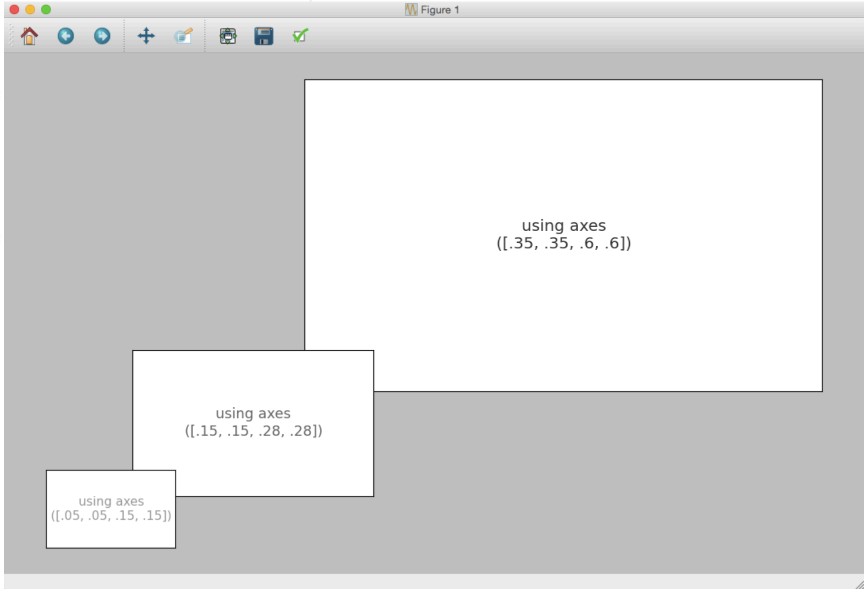
Plotting tools

- 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