A Crash Course on PYTHON Programming

I. Arregui, A. M. Ferreiro, J. A. García & Á. Leitao

Departamento de Matemáticas, Universidad de La Coruña

July, 2019

https://sites.google.com/site/crashcourseonpython/

Index

Python

- It is the most used programming language in Google

Python

- It is the most used programming language in Google
- Declared by TIOBE language of 2007, 2010 and 2018: `http://www.tiobe.com/`

Python

- It is the most used programming language in Google
- Declared by TIOBE language of 2007, 2010 and 2018: `http://www.tiobe.com/`
- Fourth most popular programming language (May, 2019): `http://www.tiobe.com/`

References

- Mark Lutz. *Learning Python*. O'Reilly, 2013
- Mark Lutz. *Programming Python*. O'Reilly, 2006
- Hans Petter Langtangen. *A Primer on Scientific Programming with Python*. Springer, 2009
- Hans Petter Langtangen. *Python Scripting for Computational Science*. Springer, 2008
- T. E. Oliphant. *Guide to NumPy*. 2006
- J. Kiusalaas. *Numerical Methods in Engineering with Python*. Cambridge University Press, 2005

- `http://www.python.org/doc`
- `http://docs.python.org/index.html`
- `http://docs.scipy.org/doc`
- `http://matplotlib.sourceforge.net/`

Documentation sites

- Official doc about `Numpy` and `Scipy`:

  http://www.scipy.org/docs.html

  Among the many links,
  - Documentation is moving to http://docs.scipy.org/doc/ where you can download:
    - **Guide to Numpy**: Travis Oliphant's book
    - **Numpy Reference Guide**: reference manual about functions, modules and objects included in `NumPy`
  - Most used `Numpy` commands:

    http://www.scipy.org/Numpy_Example_List

To practice with the contents of this course, we will need:

1. Python (versions `3.x`)
   `www.python.org`
2. MatPlotLib (for 2D graphics)
   `www.matplotlib.org`
3. SciPy: Scientific Python
   `www.scipy.org`
4. I-Python, Python advanced console
   `http://ipython.org/`

Each of these pages informs about the way to install from zero, including the dependences you need to compile

- In general, any PYTHON package includes an installation script. Thus, you just have to write:

  ```
  $ python setup.py install
  ```

  in the command line.

- Most of previous packages are available for main LINUX platforms as `.rpm` (REDHAT/FEDORA/MANDRIVA/CENTOS) or `.deb` (DEBIAN / UBUNTU)

- Installation is automatic in UBUNTU (>= 7.10) from SYNAPTIC or by `apt-get`.

- It is also possible to install all these packages in WINDOWS and MACOSX, as there exist compiled binary codes. For example:
  - ENTHOUGHT: http://www.enthought.com/products/getepd.php
  - PYTHONXY: http://www.pythonxy.com/download.php
  - ANACONDA: https://www.continuum.io/downloads

Introduction to Python

Variables and data types

Python programming

Object–oriented programming

NumPy: Numerical Python

SciPy: Scientific Python

Bidimensional graphics with MatPlotLib

Pandas

Miscelanea

Some features of PYTHON

- PYTHON is a general purpose programming language, designed by Guido van Rossum at the end of the 80's
- Clear and structured programming (for example, the tabulator is part of the language)
- Great productivity, high development velocity
- Multiple programming paradigmes: object oriented, structured, functional, ...
- Portable (Linux, Windows, Mac OSX)
- Interpreted, dynamic, strongly typed, automatic memory management
- Easy to learn
- Large *standard library*: `http://docs.python.org/library/`
- Easy to extend: link with C/C++ (`SWIG, Weave, CPython`), .NET (`IronPython`), CORBA, Java (`Jython`), FORTRAN (`f2py`), ...
- Large number of available packages

The Zen of Python

```
>>> import this
```

The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one –and preferably only one– obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

Python

Some **applications**:

- web development (Zope, Plone, Django, webpy, TurboGears, Pylons, e-mail, RSS, ...)
- access to databases (pyodbc, mysqldb, ...)
- graphical user interfaces (Tk/Tcl, WxWidgets, Qt, FLTK, Gtk, ...)
- games (PyGame, PyKyra)
- network applications: client–server (Twisted Python), ...
- graphical representation: 2D (MatPlotLib, Chaco), 3D (`VTK`, `MayaVi`), ...
- scientific computing (NumPy, SciPy)
- XML processing, HTML, ...

It works on Linux / Unix platforms, Windows, Mac, JVM (Jython), ... and there are implementations on the Nokia 60 series!

Where is Python used?

- in Google, where it is one of the three development languages (with C / C++ and Java)
- in YouTube
- in BitTorrent
- in animation: DreamWorks Animation, Pixar, Industrial Light & Magic
- the RedHat / FEdora installer (Anaconda) is written in Python
- RocksCluster is a Linux distribution for clustering, implemented on CentOSLinux, that uses Python scripts for node and users management, among others ...
- Los Alamos National Laboratory and Sandia National Laboratories have developed PyTrilinos, ParaView, ...
- Salome and Abaqus (CAD/CAE) and FEniCS (finite elements) use Python as standard script language

Some success stories: `http://www.python.org/about/success/`

Why is Python so extended?

- It is quite easy to develop **wrappers** which allow using almost every software written in C / C++ and Fortran
  - by means of the PythonC API
  - by automatic generators: SWIG, SIP, Weave, f2py

  Python is very used as a *glue*

- Almost any free software library has its corresponding wrapper, so that it can be used from Python

- Documentation is very complete
  - in the console, through function `help`
  - in the different projects webs

- Python community is very active
  - SciPy Conference, once per year in USA and Europe
  - PyCon (in USA) and EuroPython, annual conference
  - in SIAM Annual Meeting, an special session is often dedicated to the use of Python in scientific computing

Editors and command windows

Editors:

- `gedit`, `vi`, `emacs`, ...

- `eric` (`http://eric-ide.python-projects.org/`)
  both of them include directory manager, help manager, debugger,...
- `spyder`
- `Eclipse` + plugin for PYTHON
  (`http://www.eclipse.org/` and `http://pydev.org/`)

Be careful with tabulators !

- the tabulator is part of the language sintax; PYTHON uses the indentation to delimit code blocks (loops, if–loops, functions, ...)
- the standard tabulator has 4 spaces; all tabulators in the same file must be defined in the same way

Command windows (consoles):

```
$ python
$ ipython
```

The second one is more developed

Starting Python

- In Linux, we write

  `$ python`

  and we get the Python prompt: `>>>`

  Some utilities:

  ↑ or ↓: get previous commands
  Begin: we place at the beginning of the line
  End: we place at the end of the line
  `Ctrl-D` or `quit()`: exit from Python

Starting Python

Python is a **dynamically typed** language, i.e., a variable can take different values of different types in different moments

```
>>> a = 5
>>> a = "Hello, world!"
>>> b = 5.678e-3
>>> a = b + 1
>>> b = 'Bye'
```

Variable declaration is not needed

A first example

After entering the PYTHON console, we write:

```
>>> n = 100
>>> b =  range  (n+1)
>>> s =  sum  (b)
>>> b
[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, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,
73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86,
87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
>>> s
5050
```

A first example

In a different way, we can edit a file (example01.py) with the following contents;

```
# My first Python code
n = 100
b =  range  (n+1)
s =  sum  (b)                  # s = sum (range (n+1))
print
print 'The sum of ', n, ' first natural numbers is:', s
print
```

To execute it, we do:

```
$  python  example01.py
```

and we get:

```
The sum of 100 first natural numbers is:  5050
```

- If we need to use *"different"* characters —for example, ñ, á, ó (in Spanish), or č (in Czech)—, we have to include the following line in the head of the file:

```
# encoding:  utf-8
```

Help in Python

Help in command line:

```
>>>   help ()
help>
help> modules
help> keywords
help> topics
```

```
In [1]:  help ()
In [2]:  help (math)
In [3]:  help (pylab)
In [4]:  help (plot)
```

With the **Return** key or by CTRL-D, we return to the prompt.

In the I-Python window, we can get help:

- with the tabulator:

```
In[1]:   import numpy
In[2]:   c = numpy.[TABULATOR]
```

- about a loaded package or function:

```
In[3]:  help (name)
```

- about any package or function:

```
In[4]:  help ()
help> name
```

`dir (obj)`: returns a list of the atributes and methods of any object

The I–Python shell

The I–Python shell admits, among others, the usual commands of a Linux shell:

```
pwd
cd path,  cd ..
ls,  ls -al
cat a00.py
cp a00.py a01.py
mv a01.py a02.py
rm a02.py
```

The PYTHON standard library

Some modules included in the PYTHON distribution
(see http://docs.python.org/library/)

- `sys`: system-specific parameters and functions
  - `sys.argv`
  - `sys.exit`
  - `sys.stdin`, sys.stdout, sys.dtderr
- `os`: miscellaneous operating system interfaces
- `time`: time access and conversions
- `math`: mathematical functions
  - `math.sin`, `math.cos`, `math.exp`, ...
- `random`: pseudo-random numbers generations
- `string`: common string operations
- `zlib`, `gzib`, `zipfile`, `tarfile`: file compression
- `email`, `mailbox`, `ssl`, `socket`, `webbrowser`, `smtpd`, ...: e-mail tools, network, internet protocols, ...
- `audioop`, `imageop`, `wave`, ...: multimedia services
- `xml.dom`, `xml.sax`, ...: XML and HTML parses

Some scientific computing libraries

- `numpy`:
  - powerful n–dimensional arrays management
  - linear algebra basic functions
  - Fourier transforms
  - random numbers generation
- `scipy`:
  - linear systems solvers
  - sparse matrices
  - numerical integration, optimization
  - ODE's solvers
- `MatPlotLib`: two dimensional plots representation
- `tvtk`: VTK wrapper
- `mayavi2`: 2D and 3D visualization tools
- `quantlib`:

Other simple examples

Let us edit a file a00.py with the following contents:

```
import math
a =  input (' Introduce an angle:  ')
x =  math.cos  (a)
print  'Cosinus of an angle:  '
print  '      cos  (' +  str (a) + ') = ' +  str (x)
```

To execute it, we write:

```
$  python  a00.py
```

we will get:

```
Introduce an angle:
```

and if we write, for example, 3.141592654, we will obtain:

```
Cosinus of an angle:
          cos (3.141592654) = -1.0
```

Other simple examples

Another way of doing the same thing is:

```
import sys, math
import math
a =  float (sys.argv [1])
x =  math.cos  (a)
print  'Cosinus of an angle:  '
print  '      cos  (' +  str (a) + ') = ' +  str (x)
```

To execute it, we write:

```
$  python  a00.py            (we will get an error)
$  python  a00.py 3.141592654
```

and we will have:

```
Cosinus of an angle:
        cos (3.141592654) = -1.0
```

Other simple examples

And a different way to show the result is:

```
result =  '    cos (%g) = %12.5e' % (a,x)
print result
```

which gives us:

```
        cos (3.14159) = -1.00000e+00
```

It can also be executed in the I–Python console:

```
In [1]:  run  a00.py 3.141592654
```

Variables

A **variable** is a space of memory in which we store a value or set of values; each variable is given a name and can have different types (scalar, list, array, dictionary, string, ...)

PYTHON is a dynamically typed language: variables can take **different values**, **of different types**, in different moments

- the result of an expression is assigned to a variable by the assignement operator (=); e.g.,

  ```
  >>> 5 + 2; b = 3 * 4
  ```

  computes de addition (not assigned to any variable) and makes `b` take the value 12
- to continue an expression in the following line, we use the backslash (\)
- we are not obliged to initially declare the variables
- PYTHON distinguishes between capital and lower case characters
- the names of variables start by an alphabetic character

Variables and workspace

A variable can be destroyed:

```
>>> del x
```

The `type` command is used to know the type of a variable:

```
>>> type (a)
```

The workspace is the set of variables and user functions in the memory

In the I–Python shell,

- the `who` command is used to examine all the variables existing in the current instant and the imported packages
- `whos` gives some details on each variable

Data types: logical variables

They can take two values: `True` or `False`.

```
>>> m = True
>>> a = 0; b = -4;          # a and b are integer variables
>>> type (a); type (b)
>>> a = 0.; b = -4.;        # a and b are real variables
>>> type (a); type (b)

>>> xa = bool (a)   # xa is a logical variable, which value is False
>>> xb = bool (b)   # xb is a logical varialbe, which value es True

>>> c1 = (b == True)      # The result is False
                          # because b is not a logical variable
>>> c2 = (xb == False)      # The result is False
                            # because xb takes the value True
>>> c3 = (xb = False)      # We get an execution error
```

Data types: the `None` variable

Designes an empty object.

For example

```
>>> a = None
```

we create variable `a`, which contains nothing and can later be of any type (numeric, string, class object, ...); to check it, we can do:

```
>>> a is None
```

We can also create an empty list, dictionary or tuple by:

```
>>> a1 = []; a2 = { }; a3 = ()
```

```
>>> a        # Whe check the value of a
             # We get no answer because a=None
>>> a = 4.5;      # We assigne a real value
```

Data types: numerical variables

They can be:

- integer (between -2147483648 and 2147483647): `0`, `283`, `-5`
- `long`: `>>> i = 22L`
- double precission, real: `0.`, `-2.45`, `1.084E+14`
- double precission, complex: `a = 1.-4j`, `b = complex(7,-2)`

where `j` is the imaginary unit ($j = \sqrt{-1}$).

We can get the real and imaginary parts of a complex:

```
>>> b.real
>>> b.imag
```

When operating two numerical variables, the result takes the type of the "highest cathegory" operator

Data types: strings

They are enclosed in single or double quotes

```
>>> s = 'Hello, world!'
>>> t = 'And he said:  "Hello, everybody!"'
>>> print s
```

A short example, with specific functions:

```
from string import *
s1 = 'First string'
s2 = 'Second string'
s3 = "Third string"
print ' s1 = ', s1
print ' s2 = ', s2
print ' s3 = ', s3
```

```
s01 = s1 + s2 + s3;  print s01
s02 = s1 + ' ' + s2 + ' ' + s3
print s02
print split (s01, sep=' ')
print count (s01,'st')
print capitalize (s02)
print lower (s01)
print replace (s02,'t','p')
```

Data types

```
>>> a = 'Welcome message'
>>> b = 4; c = 130L; d = 4.56789; m = 4.2-8.424j

>>> cs = str (c)      # '130'
>>> ds = str (d)      # '4.56789'
>>> ms = str (m)      # '(4.2-8.424j)'

>>> bd = float (b)      # 4.0
                  #  can be applied to strings of numerical characters,
                  #  integers and complexes with no imaginary part

>>> di = int (d)      # 4
                  #  returns the nearest integer towards zero;
                  #  can be applied to strings of numerical characters,
                  #  and complexes with no imaginary part
>>> bc = complex (b)      # 4+0j
>>> dc = complex (d)      # 4.56789+0j
```

Tuples and lists

Tuples and **lists** allow the storing of an arbitrary sequence of objects

The stored objects may be of different types: string, numbers, other lists or tuples, ...

We access each element by an index, which indicates the position of the element in the list or tuple

- indexes always start in *zero*

Difference between tuples and lists:

- A tuple can never be modified
- In a list, we can change, add and delete elements

Tuples

A tuple is an arbitrary sequence of objects, enclosed in **parentheses** and separated by *commas*, ( )

The numbering of indices starts at **cero**

```
>>> x = ()                  # Empty tuple
>>> x = (2,)                # Tuple of one only element
>>> x = (1, 4.3, 'hello', (-1,-2))
>>> x[0]                    # We get the first component
1
>>> x[3], x[3][1]
(-1, -2), -2
>>> x[-2]                   # Starting from the end
'hello'
>>> y = 2,3,4               # Parentheses can be omitted
>>> z = 2,                  # Tuple of an only element
```

`len (x)`: length of a tuple

`max (x)`, `min (x)`: maximum/minimum of a tuple

`tuple (seq)`: transforms `seq` in a tuple

```
>>> tuple ('hello')
('h','e','l','l','o')
```

Lists

A list is an arbitrary sequence of objects, enclosed in **brackets** and separated by *commas*, [ ]
The numbering of indices starts at **cero**

```
>>> x = []          # Empty list
>>> x = [1,4.3,'hello',[-1,-2],'finger',math.sin,[-0.4,-23.,45]]
>>> x[0]
1
>>> x [3], x [3][0]
[-1, -2], -1
>>> x[-3]           # Starting by the end
'finger'
>>> x [2:4]         # Returns a (sub)list: [x[2], x[3]]
['hello', [-1,-2]]
>>> x[4:]
['finger', <built-in function sin>, [-0.4,-23,45]]
>>> x[7]
Traceback (most recent call last):
File "<input>", line 1, in ?
IndexError:  list index out of range
```

Useful list functions

`range ((first,)last(,step))`: creates a list of integers, since `first` until `last` (*not included!*) with step `step`.

- if `first` is not given, it starts in *zero*
- if `step` is not given, the step is *one*

```
>>> range (-1, 5, 1)
[-1, 0, 1, 2, 3, 4]
>>> range (-10, 10, 2)
[-10, -8, -6, -4, -2, 0, 2, 4, 6, 8]
```

`len(x)`, `max(x)`, `min(x)`

```
>>> x = [-120, 34, 1, 0, -3]
>>> len (x)      # Number of elements of x
5
>>> max (x)
34
>>> min (x)
-120
```

Managing lists

`map(func,xlist)`: applies function `func` to every element of `xlist`

```
>>> x = ['Felix','Graziana','ROBERTA','luIS']
>>> map (string.lower,x)      # writes all strings in lower case
['felix','graziana','roberta','luis']
```

The following methods can be applied to any object of type `list`:

- `append(obj)`: adds an element `(obj)` at the end of the list
- `extend(obj)`: adds the elements of list `obj` to the current list

```
>>> z = [1,2]
>>> z.append (['dog','cat'])
>>> z
[1,2,['dog','cat']]
```

```
>>> z.extend ([20,'hello',-40])
>>> z
[1,2,['dog','cat'],20,'hello',-40]
>>> z + [20,'hello',-40]
```

Managing lists

`x.insert(j,obj)`: inserts an element `obj` in position `j` of the list
`x.remove(obj)`: finds the first element coincident with con `(obj)` and removes it from the list
`x.pop(j)`: if an index (`j`) is given, it removes the element in its position; if not, it removes the last element
`x.reverse()`: writes the list in inverse order
`x.sort()`: sorts the list

```
>>> x = [1,'hello',[2,3.4],'good',[-1,0,1],'bye']
>>> x.remove ('hello')
>>> x
[1,[2,3.4],'good',[-1,0,1],'bye']
>>> x.pop (1)
[2,3.4]
>>> x
[1,'good',[-1,0,1],'bye']
```

```
>>> x.insert (2,'text')
>>> x
[1,'good','text',[-1,0,1],'bye']
>>> x.pop ()
'bye'
>>> x
[1,'good','text',[-1,0,1]]
```

Managing lists

We can create a list with all elements equal:

```
>>> n = 5
>>> x = 0.25
>>> a = [x]*n
>>> a
[0.25, 0.25, 0.25, 0.25, 0.25]
```

A string can be considered as a list:

```
>>> s = 'Hello, world'
>>> s[0]
'H'
>>> s[1]
'e'
>>> s[-1]
'd'
```

We can convert an object into a list: `>>> list(obj)`

```
>>> x=(1,2,3); y=list(x)
```

Exercise 1

- We create two empty lists to store names and identity card numbers:

```
>>> names = []
>>> idnumbers = []
```

- We add data to each list:

```
>>> idnumbers.append ('33807659D');
>>> idnumbers.append ('32233322K')
>>> names.append ('Franz Schubert')
>>> names.append (r'Claudio Monteverdi')
```

- We create a list with the two previous lists:

```
>>> data = [idnumbers, names]
```

- We access the data:

```
>>> data[0]
>>> data[1]
>>> data[0][1]
```

Exercise 2

Create a list with the first terms of Fibonacci sequence

```
>>> a = [0, 1]
>>> a.append (a[0] + a[1])
>>> a
>>> a.append (a[1] + a[2])
>>> a.append (a[2] + a[3])
>>> a.append (a[3] + a[4])

>>> a.append (a[-2] + a[-1])
>>> a.append (a[-2] + a[-1])
>>> a.append (a[-2] + a[-1])
>>> a.append (a[-2] + a[-1])
```

Exercise 3

For a 2–D mesh, we want to create:

- a list of nodes, with the two coordinates of each
- a list of elements, with the indices of the vertices

- Coordinates:

```
>>> nodes = [[0., 0.], [1., 0.], [1., 1.], \
        [0., 1.], [2., 0.], [2., 1.]]
>>> nodes [0]     # [0., 0.]
>>> nodes [2]     # [1., 1.]
>>> nodes [5]     # [2., 1.]
```

- Elements:

```
>>> conec = [[0, 1, 2], [0, 2, 3], [1, 4, 5], [1, 5, 2]]
>>> conec [2]
>>> conec [3]
>>> nodes [conec[0][2]]     # [1., 1.]
```

Exercise 3

Another possibility:

- Coordinates:

```
>>> nodes = [(0., 0.), (1., 0.), (1., 1.), \
        (0., 1.), (2., 0.), (2., 1.)]
>>> nodes [0]     # (0., 0.)
>>> nodes [2]     # (1., 1.)
>>> nodes [5]     # (2., 1.)
```

- Elements:

```
>>> conec = [[0, 1, 2], [0, 2, 3], [1, 4, 5], [1, 5, 2]]
>>> conec [2]
>>> conec [3]
>>> nodes [conec[0][2]]     # (1., 1.)
```

Dictionaries

A dictionary allows to store an arbitrary sequence of objects

- We access its elements through **keys**, which are variables of any type (not only integers)
- The elements are sorted by keys alphabetical order

Exemple:

```
>>> x = { key01:  'first element' }
```

or:

```
>>> x = { }        # empty dictionary
>>> x [key01] = 'first element'
>>> x [key02] = 25.50


>>> x
{ key01:  'first element', key02:  25.5 }
```

Dictionaries

Other example:

```
>>> clients = {'smith':  ['Adam Smith',38,'44000111D'],
              'roberts':  ['Mary Roberts',17,'33221998L']}
>>> clients['roberts']
['Mary Roberts',17,'33221998L']
```

How to add, modify and delete elements?

```
>>> clients ['white'] = ['Charles White',23,'44555111L'] # add
>>> clients ['smith'] = ['Adam Smith',29,'44000112D'] # modify
>>> del clients ['roberts']
>>> clients
```

`>>> namedict.has_key (namekey)`: returns `True` if the dictionary `namedict` has the key `namekey`; otherwise, it returns `False`

Dictionaries

- `len (ndic)`: number of elements of the dictionary `ndic`
- `ndic.keys ()`: returns a list with the keys
- `ndic.values ()`: returns a list with the values
- `ndic.items ()`: returns the contents in tuples
- `ndic.update (ndic2)`: adds the elements of one dict. to another
- `ndic.clear ()`: deletes all the elements of ndic

For example,

```
>>> clients = {'wagner':['R. Wagner',19],
           'byrd':['W. Byrd',45], 'white':['C. White',23]}
>>> clients.keys()
['wagner','byrd','white']
>>> len (clients)
3
>>> clients.clear()
>>> clients
{ }
```

Exercise

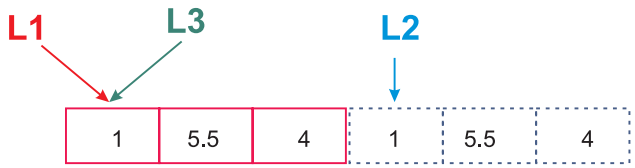
In the previous 2–D mesh, make a dictionary of the elements

```
>>> elem = {  }
>>> elem[0] = [0, 1, 2]
>>> elem[1] = [0, 2, 3]
>>> elem[2] = [1, 4, 5]
>>> elem[3] = [1, 5, 2]

>>> elem
>>> elem.keys ()
>>> elem.values ()
>>> elem.items ()
```

Value copy

A value copy of a list/tuple/dictionary is a different object with the same content than the original one; the new object points to **a different memory position**

```
>>> L1 = [1,  5.5,  4]
>>> L2 = L1[:]          # A copy of the list
>>> L2 is L1
False
>>> L2.append (100)     # If we modify L2, then L1 doesn't change
```



Another way of doing the same process is by means of the `copy` function:

```
>>> import copy
>>> L1 = {'a':1,  'b':22,  'c':4}
>>> L2 = copy.copy (L1)
>>> L2 is L1
False
>>> L1['d'] = 88     # If we modify L1, then L2 is not affected
```

Reference copy

By a reference copy, we create an object that **points to the same memory position** of the original object

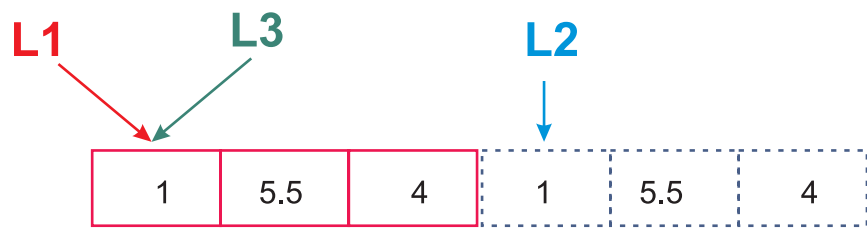
```
>>> L1 = [1, 5.5, 4]
>>> L3 = L1      # Creates a new variable, that points to L1
>>> L3 is L1
True
```

If we modify `L3`, then we also modify `L1`:

```
>>> L3.append (100)
>>> L1
[1,  5.5,  4,  100]
```

If we delete `L1`, then `L3` is not deleted:

```
>>> del L1;   L1
Traceback (most recent call last):
File "<input>", line 1, in ?
NameError:  name 'L1' is not defined
>>> L3
[1,  5.5,  4,  100]
```



Introduction to Python

Variables and data types

Python programming

- Control sentences
- Functions
- Modules
- The standard library
- Input / output
- Exceptions

Object–oriented programming

NumPy: Numerical Python

SciPy: Scientific Python

Bidimensional graphics with MatPlotLib

PYTHON programming

Names of the files: `namefile.py`

They contain sequences of commands and orders

Script

- they are executed by typing:

```
$ python namefile.py
```

or

```
In[1] run namefile.py
```

Functions

- its structure is:

```
def namefunc (x1,x2,...):
        ⋮
    return [a,b,c]
```

- they are executed by a call from a script, another function or the command window (PYTHON or I–PYTHON shell)

They are recursive

Compiling and linking are not necessary

Bifurcations (if)

```
if condition:
    sentence(s)
```

```
if condition1:
    sentence(s)
elif condition2:
    sentence(s)
```

```
if condition1:
    sentence(s)
elif condition2:
    sentence(s)
elif ... :
    ...
else:
    sentence(s)
```

- Conditions can be boolean, scalar, vectorial or matricial; in this case, vectors and matrices must have equal dimensions
- They can be imbricated
- Only a block of sentences is executed
- Operators:

  `<` `<=` `>` `>=` `==` `!=` `or` `and` `not`

- **IMPORTANT:** Indentation must be done with the tabulator !!!

Bifurcations

The result of each *condition* will be `True` or `False`.

In particular, if we do

```
if var
```

where `var` contains a variable, the result will be:

- `False`, if `var` is:
  - the `None` variable
  - a numerical variable, with value *zero*
  - a boolean variable, with value `False`
  - an empty tuple, list or dictionary
- `True`, otherwise

Loops

```
for i in range (i1,i2,step):
    sentence(s)
```

```
while (condition):
    sentence(s)
```

- if `step` is *one*, it can be omitted
- if `i1` is *zero* and the step is *one*, they both can be omitted

```
for item in ObjIt:
    sentence(s)
```

where `ObjIt` is an iterable object

Indentation must be done with the tabulator!

```
1 import math
2 methods = [math.sin, math.cos, math.tan, math.exp]
3 for k in methods:
4     print k.__name__, '(pi)= ', k (math.pi)    # Python 2.x
5     # print (k.__name__, '(pi)= ', k (math.pi)) # Python 3.x
```

Loops

We can use several lists simultaneously:

```
for x, y, z in zip (xlist,ylist,zlist):
    print x, y, z                    # Python 2.x
    # print (x, y, z)                # Python 3.x
```

- In this case, the number of iterations will be the dimension of the shortest list

Loops can be imbricated:

```
for i in range (i1,i2,step1):
    for j in range (j1,j2,step2):
        print a[i][j]                # Python x.x
        # print (a[i][j])            # Python 3.x
```

Two important commands:

`break`: quits the inner loop

`continue`: starts a new iteration of the same loop

switch

PYTHON has not the command `switch`, but we can do something similar by a dictionary:

```
1  from math import *
2
3  num = pi
4  menu = {'1': cos, '2': sin, '3': exp}
5  choice = input ("Choose an option [1 / 2 / 3]")
6  val = menu [choice] (num)
7  print '%s (pi) = %f' % (menu[choice].__name__, val)
8                  # Python 2.x
9  # print ('%s (pi) = %f' % (menu[choice].__name__, val))
10                 # Python 3.x
```

Exercises

Write a PYTHON code ("`days.py`") which read a date (day, month and year), check if it is correct and write it in the screen.

```
import types
y = input ('Write the year:     ')
if (y <= 0 or type (y) != types.IntType):
    print 'The year is not correct ... '       # Python 2.x
    # print ('The year is not correct ... ')  # Python 3.x
```

or:

```
import types
y = -1
while (y <= 0 or type (y) != types.IntType):
    y = input ('Write the year:     ')
print 'The date is:   \%2d.\%2d.\%4d'  \% (d,m,y)
# print ('The date is:   \%2d.\%2d.\%4d'  \% (d,m,y))
```

Exercises

In the previous code, compute the sum of the days of the months previous to the current one.

```
aux = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
total = 0
for i in range (0,m-1):
    total = total + aux (i)
```

or:

```
aux = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
total = sum (aux(0:m-1))
```

Exercises

In the previous code, determine if the current year is a leap–year and modify the code to get the correct result.

```
b = 0
if (y % 4 == 0):
    b = 1
    if (y % 100 == 0 and y % 400 != 0):
        b = 0
aux[1] += b          # aux[1] = aux[1] + b
```

Functions

Their structure is:

```
def f (x1,x2,x3):
    ...
    y1 = ...; y2 = ...;
    return [y1, y2]
```

They can return any type of variable: string, scalar, list, tuple, array, class object, class instance, ...

They admit a variable number of arguments

```
y = f (a, b, c, d, p, q, r, s)
def f (x1, x2, x3, *x4):
    if len (x4) == 4:
```

Default arguments are allowed:

```
def f (x1, x2, x3=1.0, x4=None):
```

Functions

We can give, as argument of a function, the name of another function

```
import math as mt
def f (x):
  return x**2

def g (x):
  return x**3

def h (x):
  return mt.sqrt (abs(x))

def func (x,f):
  map (f,x)
```

```
a = range (-5,6)
opt = 1
if opt == 1:
  print func (a,f)
elif opt == 2:
  print func (a,g)
else:
  print func (a,h)
```

Functions: recursivity

A function can be called by itself

```
1 import types
2 def fact (n):
3     if type (n) != types.IntType:
4         return -1
5     else:
6         if n == 0:
7             return 1
8         else:
9             return n*fact (n-1)
```

`lambda` functions

The `lambda` expression is used to define simple functions *inline*

```
lambda arg1, arg2, ..., argN: expression
```

```
>>> f = lambda x, y: x*y
>>> f (2,3)
6
>>> import math
>>> g = lambda x, k = 1.0, a = 1.: k * math.exp (a*x)
>>> g (2, a=-1)
0.135335283236612 7
```

`lambda` functions

- `lambda` functions are often used to codify *jump tables*, which are actions stored in lists or dictionaries, to be executed when needed

```
1  import math
2
3  LF = [lambda x : math.sin (x) ,
4        lambda x : math.cos (x) ,
5        lambda x : math.exp (x / math.pi)]
6
7  for f in LF:
8    print f (math.pi)          # Python 2.x
9    # print (f (math.pi))      # Python 3.x
10
11 print LF [1](2.* math.pi)
12 # print (LF [1](2.* math.pi))
```

Functional programming tools

- `map (func, seq)`: applies a function to a sequence of data

```
# In Python 2.x:
>>> map (lambda x : x**2 , [1, 2, 3])
# In Python 3.x:
# >>> list (map (lambda x : x**2, [1, 2, 3]))
[1, 4, 9]
```

- `filter (func, seq)`: filters the elements of an iterable object

```
# In Python 2.x:
>>> filter (lambda x : x > 0, [ -1, 3, -4, 5])
# In Python 3.x:
# >>> list (filter (lambda x : x > 0, [ -1, 3, -4, 5]))
[3, 5]
```

- `reduce (func, seq)`: applies a two arguments function, `func(x,y)`, in a cumulative way to the elements of an iterable object, `seq`, from left to right, to reduce it to an only value

```
>>> from functools import reduce   # Only in Python 3.x
>>> reduce (lambda x, y : x+y, [ -2, -3, -4, -5])
-14              # computes : ((-2+(-3))+(-4))+(-5)
>>> reduce (lambda x, y : x*y, [ -2, -3, -4, -5])
120              # computes : ((-2*(-3))*(-4))*(-5)
```

Working with our modules

In file `norms.py` we implement three functions that compute the one–norm, two–norm and infinity–norm of a list of numbers $L = [x_0, \ldots, x_n]$:

$$||L||_1 = \sum_{i=0}^{n} |x_i|, \qquad ||L||_\infty = \max_{i=0,\ldots,n} \{|x_i|\}, \qquad ||L||_2 = \sqrt{\sum_{i=0}^{n} (x_i)^2}$$

```
1  # File norms.py
2
3  # encoding: utf-8
4
5  def n1 (L):                # L is a list
6      return reduce (lambda x, y : abs (x) + abs (y), L)
7
8  def ninf (L):
9      return max (map (abs, L))
10
11 import math
12 def n2 (L):
13     return math.sqrt (sum (map (lambda x : x**2, L)))
```

Working with our modules

- In order to instantiate the methods of the `norms.py` module from the file `main.py` (both lying in the same directory), we must first "import" the module: `import norms`

```
# File: main.py

import norms

L = [-1, 2, 3, -5]
print ' L = ', L
print ' norm 1 (L) = ', norms.n1 (L)
print ' norm 2 (L) = ', norms.n2 (L)
print ' norm Inf (L) = ', norms.ninf (L)
```

- The sentence `from ModuleNameimport ModuleAttribute` allows to import the indicated function (or functions):

```
from norms import ninf
print ninf ([-2, -20, 10, 50])
print n1 ([-2, -20, 10, 50])          # Error!
```

```
from norms import ninf , n1 , n2
print ninf ([-2, -20, 10, 50])
print n1 ([-2, -20, 10, 50])
```

PYTHONPATH

The place where PYTHON stores the path of the modules that can be imported is the environment variable `PYTHONPATH`

We can modify it in several ways:

- In bash, for example, we can add the following lines to the file `.bashrc`:

```
export PYTHONPATH=$PYTHONPATH:/mydir1:/mydir2
```

- In the PYTHON code, the simplest way is:

```
import sys
sys.path.append ("/home/me/mydir1")
sys.path.append ("/home/me/mydir2")
```

The standard library: `os` module

`os` module contains many functions for files and processes managing.

`os.path`: paths and files managing

```
os.mkdir ('subdir44')
os.getcwd ()      # the directory where we are
os.chdir ('subdir44')
os.chdir (os.environ['HOME'])
os.rename ('fic01.py','fic02.c')
os.remove ('fic02.c')
os.listdir (os.curdir)    # files in the directory
os.listdir ('/home/me/dir2')
```

operating system functions: `os.kill`, `os.execv`, ...

The standard library: `sys` module

- `sys.argv`: list of arguments received by a script in the command window
- `sys.exit`: exit from PYTHON
- `sys.stdin`, `sys.stdout`, `sys.stderr`

```
1  import sys
2
3  # Data given by keypad
4  data = sys.argv[:]
5
6  print 'data: ', data        # Python 2.x
7  # print ('data: ', data)   # Python 3.x
8
9  print ("We have passed %d entry arguments to %s code: " % (
       len(sys.argv)-1, sys.argv[0]))
10 for arg in sys.argv[1:]:
11     print " %s" % arg
12     # print (" %s" % arg)
```

The standard library: time measure

PYTHON provides the data and time in three different formats:

- as a tuple: year–month–day–hour–minute–second–day of the week—day of the year—X (*tup*)
- as a string (*str*)
- as the total number of seconds from the origin (*sec*)

`time ()`: current instant

`clock ()`: elapsed time from the beginning of the execution

`sleep (n)`: pause, `n` seconds

`gmtime ()`: GMT hour

`localtime ()`: local hour

`asctime` *(tup)*: transforms the tuple in string

`strftime` *(tup)*: transforms the tuple in string

`mktime` *(tup)*: transforms the tuple in seconds

`ctime` *(sec)*: transforms the seconds in string

`strptime` *(str)*: transforms the string in tuple

Interactive I/O

Data reading by keyboard in PYTHON 2.X:

`x = raw_input ('message')`

`'message'` is written in the screen and waits until a data is provided; it is stored in `x` variable as a string.

We cannot operator with `x`

```
>>> x = raw_input ('Introduce a number:  ')
124.5
```

`y = x**2` **is not possible**, because `x='124.5'`

We should do: `y = float(x)**2`

`x = input ('message')`

`'message'` is written in the screen and waits until a data is provided; PYTHON tries to evaluate it in order to store it in a correct `x` variable

This way, we can operate with `x`:

```
>>> x = input ('Introduce a number:  ')
124.5
>>> y=x**2
```

Interactive I/O

Data reading by keyboard in PYTHON 3.X:

```
x = input ('message')
```

'message' is written in the screen and waits until a data is provided; it is stored in x variable as a string.

We cannot operate with x

```
>>> x = input ('Introduce a number:  ')
124.5
```

y = x**2 **is not possible**, because x='124.5'

We should do: y = float(x)**2

Interactive I/O

Non formatted data writing (in PYTHON 2.X):

```
>>> print 'message', var1, var2, ..., vark
```

`message` and the values of `var1, var2, ..., vark` are written in the screen
Variables are separated by a blank:

```
>>> nm = 'Andrew'
>>> age = 45
>>> print 'Name, age = ' , nm, age
Name, age = Andrew 45
>>> print 'Name = ', nm, ' age = ', age
Name = Andrew age = 45
```

Interactive I/O

Non formatted data writing (in PYTHON 3.X):

```
>>> print ('message', var1, var2, ..., vark)
```

`message` and the values of `var1, var2, ..., vark` are written in the screen
Variables are separated by a blank:

```
>>> nm = 'Andrew'
>>> age = 45
>>> print ('Name, age = ' , nm, age)
Name, age = Andrew 45
>>> print ('Name = ', nm, ' age = ', age)
Name = Andrew age = 45
```

Interactive I/O

Formatted data writing:

```
print 'Msg1 = %format1, Msg2 = %format2' % (var1,var2)
#print ('Msg1 = %format1, Msg2 = %format2' % (var1,var2))
```

where `format` describes the way in which we want to representate each variable:

- `%i` or `%d`: integer
- `%f`: float in decimal format
- `%e`: float in exponencial format
- `%g`: non leading zeros are removed
- `%s`: string

```
>>> name = 'Joe'
>>> s = 50.5
>>> print  '%s earns %7.2f euros a month' %  (name, s*30.)
>>> print  ('%s earns %7.2f euros a month' %  (name, s*30.))
Joe earns 1515.00 euros a month
```

Raw strings

Sometimes, we need to store strings with special characters; for example:

```
path1 = 'C:\\\\Windows\\\\Temp'
print ' In Windows, you can find the file in ', path1
```

Other special characters are `\n` (break line), `\t` (tabulator), ...,

We can manage these strings as **raw strings**:

```
path1 = r'C:\\Windows\\Temp'
```

Text files reading

- File identification: `idf = file (pathfile,'r')`
  If the file doesn't exist, PYTHON returns an error

```
>>> idf1 = file ('file1.txt','r')
>>> idf2 = file ('/home/arregui/python/file2.dat', 'r')
```

- One line reading:

```
>>> text = idf.readline ()
```

  in variable `text` we store the first line of the file `'file1.txt'`, plus the line break (`\n`)

```
>>> idf.readline ()
'1 2 3.456 8.001 -99.01\n'
```

- Whole file reading:

```
>>> text = idf.read ()
```

  Returns a string with the contents of the whole file; por example

```
>>> idf.read ()
'F1 F3 F5\nReading a file\n'
```

Text files reading

- We can start reading at any point of a file; for example, at the beginning:

  ```
  >>> idf.seek (0)
  ```

  In general, `seek (n)` locates the pointer `n` bytes away from the beginning
- We can read all the lines of the file:

  ```
  >>> idf.readlines()
  ```

  and we get a list of strings

  ```
  >>> lines = idf.readlines()
  >>> lines[1]
  'F1 F3 F5\n'
  ```
- To close the file, we do:

  ```
  >>> idf.close()
  ```

  and we can delete the file identified by `idf`:

  ```
  >>> del idf
  ```

Text files writing

Creation of the `file` object:

```
idf = file (filepath,'w')
```

If the file does not exist, it is created

For adding text, we do:

```
idf.write (string)
```

```
>>> idf.write ('First line of the file \n')
```

If we write again, the new string is added to the file

```
>>> idf.write (""" I am writing
a multiline.  This is a test.
End """)
```

To close de file, we do:

```
idf.close ()
```

If we open the file, (from the LINUX shell, before it is closed from PYTHON, we will see it empty

Numerical files reading and writing

If all the lines have the same number of columns, we can use the `loadtxt` and `savetxt` commands, availables in NumPy library

`loadtxt` reads a file and returns an $m \times n$ `array` (where m and n are the numbers of lines and columns, respectively, of the file)

```
A = loadtxt (path[,delimiter=None]))
```

```
1 from numpy import loadtxt, savetxt, array
2 filename = 'matrix.txt'
3 A = loadtxt (filename)
```

`savetxt` saves a matrix in a file:

```
savetxt (filename, A (,fmt='%format', delimiter='symb'))
```

where:

`format` indicates the format in which we want to represent the variable: `%i`, `%d`, `%f`, `%e`, `%g`

`symb` indicates the character that will separate the columns in each line; by default, it is a blank space

```
1 A = array ([[2.3, -4.4, 0], [-20, 4.4, -1]])
2 savetxt ('matrix2.txt', A, fmt='%7.2f', delimiter=',')
```

Numerical files reading and writing

If the number of columns is not the same in every row, we have to read each line:

```
from string import split
k = 1
while (k>0):
      s = split  (f.readline())
      if (s):
            a = map (float,s)
      else:
            k = -1
```

where:

- `split(s(,';'))`: method of the `string` module that splits a string in words, separated by blank spaces (by default) or by the provided symbol
- `map`: function that applies another function to every element of a list

Exceptions

We can distinguish between two kinds of errors:

- Syntax errors

```
>>> while True print ('hello')
File "<stdin>", line 1
   while True print ('hello')
                    ^
SyntaxError: invalid syntax
```

  In this example, the colon (":") is missing

- Exceptions

```
>>> 5 / 0
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Exceptions can be captured:

```
try:
   sentences
except [TypeException]:
   sentences
```

Exceptions (example I)

```
>>> 3./ 0.
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError:  float division
```

Exceptions (example I)

```
>>> 3./ 0.
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError:  float division
```

Control of the error:

```
a = 5.0
b = 0.0
try:
    res = a / b
except ZeroDivisionError:
    print ('Division by zero')
```

Exceptions (example II)

```
>>> f = open ('myfile.txt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IOError:  [Errno 2] No such file or directory:  'myfile.txt'
```

Exceptions (example II)

```
>>> f = open ('myfile.txt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IOError:  [Errno 2] No such file or directory:  'myfile.txt'
```

Control of the error:

```
import sys
try:
    f = open ('myfile.txt')
    s = f.readline ()
    i = int (s.strip ())
except IOError,(errno,strerror):
    print ("I/O error (%s):  %s"  % (errno,strerror))
except ValueError:
    print ("The data cannot be converted to integer")
except:
    print ("Error:", sys.exc_info()[0])
```

Exercises

1. Write a PYTHON code that:
   (a) reads the components of a vector
      - from the screen
      - from a file
   (b) computes its harmonique average α:

$$\alpha^{-1} = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{a_i}$$

   (c) computes its norms:

$$\|a\|_2 = \left(\sum_{i=1}^{n}|a_i|^2\right)^{1/2}$$

$$\|a\|_\infty = \max_{i=1,\ldots,n}|a_i|$$

Exercises

2. Solve the equation:

$$f(x) = x^5 - 3.8x^3 - 4 = 0 \qquad x \in (-2, 2.5)$$

by the bissection method: a_0, b_0 given,

$$x_k = \frac{1}{2}(a_k + b_k)$$

$$\text{si} \;\; f(x_k)f(a_k) < 0 \quad \Longrightarrow \quad \begin{cases} a_{k+1} = a_k \\ b_{k+1} = x_k \end{cases}$$

$$\text{si} \;\; f(x_k)f(b_k) < 0 \quad \Longrightarrow \quad \begin{cases} a_{k+1} = x_k \\ b_{k+1} = b_k \end{cases}$$

Exercises

3. Consider the (convergent) numerical series: $S = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{\sqrt{1+4n}}$.

   Write a PYTHON code that computes its sum with an error less than a given parameter.

4. Write a PYTHON code that reads a date (day, month, year), check if it is a correct date and computes the days since January 1st of that year.

5. Compute the age of a person in days

   Which are leap–years? Multiples of 4, unless multiples of 100 which are not multiples of 400

Exercises

6. Write a PYTHON code that generates a dictionary with:
   - titles and production year of films
   - names of directors
   - names of actors

   and show the information of the films sorted by:
   - key
   - date
   - director name

Object–oriented programming

Object–Oriented Programming (OOP) is a programming paradigm that tries to represent entities (objects) grouping data and methods that describe their features and behavior

In OPP, programs are organized "as in the real life", by means of classes and objects that interact with them
For example, let us consider the points $(1, 2)$, $(3, 4)$ y $(5, 7)$

- all of them are different
- but they have all two coordinates, with different values
- and we can apply the same methods (functions) to all of them

OOP advantages

- The code is much simpler to write, maintain, re-use and extend
- Creating more complex codes is easier
- The implementation of the codes is related to the "organization of the real world"

Object–oriented programming

- A **class** is a sort of *template*, where attributes and methods are defined
- Classes are not manipulated directly; they are used to define new types (**instancias**) of the class or **objects**

  For example, a class could be the plan of a house; it is generic. The instances are the different houses that can be built from the plan
- A class **method** is a function that can be applied to any instance of the class. A class **attribute** is a variable associated to each instance (a property)
- A method and/or attribute is **public** if it is accesible at any point of the code, and it is private if it is only accesible by the methods of the class

Object–oriented programming

OOP is posible in Python

- Creation of a class: `class ClassName:`
- A class has an init method, which is used to create instances of the class

```
class ClassName:
        def __init__ (self,args):
        ...
```

- By convention, the first argument of a class method is `self`, which is a pointer to the class; it is equivalent to `this` in C++ or Java
  - It is not a reserved word, but it is used by convention
- Although `self` must be the first argument of a class method, we do not need to include it as Python invokes it automatically

Instances of a class

- For creating an object, or instance, of a class we have just to invoke the class as if it was a function, giving it the arguments of the `__init__` method

- The returned value is the created instance

```
1 class point:
2     def __init__ (self, x, y):
3         self.x = x            # attribute
4         self.y = y            # attribute
5
6 p1 = point (0.0, 1.0)
7 p2 = point (3.0, 5.0)
8 print ('p1 is p2: ', p1 is p2)
```

Operator overloading

- PYTHON does not allow the method overloading (C++ and JAVA do)
- But operator overloading is allowed: we can redefine operators as $+$, $-$, $*$, $/$, `print`, for example

Method	Overload	How to call it
`__init__`	Initializer	`A = ClassName ()`
`__del__`	Destructor	`del A`
`__str__`	Printing	`print (A), str(A)`
`__call__`	Evaluates the object in the args	`A(args)`
`__add__`	`+` operator	`A+B, A += B`
`__sub__`	`-` operator	`A-B, A -= B`
`__mul__`	`*` operator	`A*B, A *= B`
`__div__`	`\` operator	`A\B, A\=B`
`__pow__`	`**` operator	`A**B, pow(A,B)`
`__eq__`	`==` operator	`A == B`
`__lt__`	$<$ operator	`A < B`
`__comp__`	Comparison	`A==B, A<=B,A>=B, A<B,A>B, A!=B`
`__or__`	`|` (OR) operator	`A|B`
`__getitem__`	Gets the value of `k` index	`A[k]`
`__setitem__`	Assigns a value to `k` index	`A[k] = value`
`__getattr__`	Gets an attribute	`A.attribute`
`__setattr__`	Sets the value of an attribute	`A.attribute = value`

Example: a `point` class

```
1  class point:
2      def __init__ (self, x=0, y=0):
3          self.x = x
4          self.y = y
5
6      def print (self):
7          print ' Point: ( ', self.x,' ,  ', self.y,' )'
8          # print (' Point: ( ', self.x,' ,  ', self.y,' )')
9
10     def modify_coord (self, newx, newy):
11         self.x = newx
12         self.y = newy
13
14     def move (self, u, v):
15         self.x += u
16         self.y += v
17
18     def newpoint_move (self, u, v):
19         p2 = punto ()
20         p2.x = self.x + u
21         p2.y = self.y + v
22         return p2
```

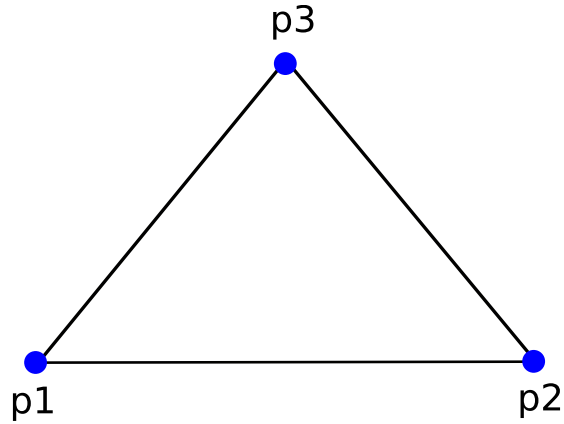
Example: a `point` class

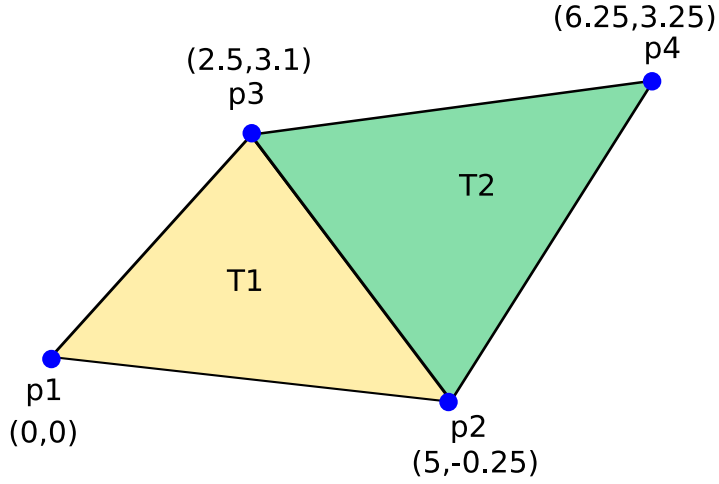
```
    def __add__ (self, p):
        self.x += p.x
        self.y += p.y
        return self

p0 = point ()
p1 = point (0., 1.)
p2 = point (3, 5)

p1.modify_coord (2.0, 6.0)
p1.print ()

p1.print ()
print (' p2: (%f, %f)' % p2.x, p2.y)
p3 = p1 + p2
p3.print ()
```

Example: a `triangle` class

```
1  import point
2
3  class triangle:
4
5      def __init__ (self, p1, p2, p3):
6          self.vertices = [p1, p2, p3]
7
8      def print (self):
9          # Writes the points coordinates
10         print '---  Vertices of the triangle ---'
11         # print ('---  Vertices of the triangle ---')
12         for k in self.vertices:
13             k.print ()
14
15     def mod_pointcoord (self, ind, x, y):
16         self.vertices [ind].modify_coord (x,y)
17
18     def surface (self):
19         v = self.vertices
20         x1 = v [1].x - v [0].x
21         y1 = v [1].y - v [0].y
22         x2 = v [2].x - v [0].x
23         y2 = v [2].y - v [0].y
24
25         return abs (x1*y2-x2*y1)
```

Example: a `triangle` class





Example: a `triangle` class

```
1  import point
2  import triangle
3
4  p1 = point (0.0,0.0); p2 = point (5,0.1)
5  p3 = point (2.5,3.5); p4 = point (7.2,1.5)
6
7  t1 = triangle (p1, p2, p3)
8  t2 = triangle (p2, p4, p3)
9
10 for t in [t1,t2]:
11     t.print ()
12     print 'Triangle surface: ', t.surface ()
13     # print ('Triangle surface: ', t.surface ())
```

Example: a `triangle` class

```
# If we modify the coordinates of p2, changes affect to its
    vertices
# as they have the same memory direction

p2.modify_coord (4.5, -0.25)
for t in [t1,t2]:
    t.print ()

t1.mod_pointcoord (2, 2.65, 10)

for t in [t1,t2]:
    t.print ()
```

Object–oriented programming: heritage

Heritage is one of the most important features of OOP

- New classes can be created from an existing class, from which they inherit attributes and methods, that can be adapted and/or extended
  For example:
  - A car, a plane, a train and a ship are all vehicles which have common attributes and methods (number of passengers, maximum speed, turn off, turn on, ...); each of them can have its particular attributes and methods (number of wheels, number of wings, ...)
    We can consider a `vehicle` class, and a `car`, `plane`, `train` and `ship` subclasses
  - A triangle and a rectangle are both polygons
- The new class is known as derived class, child class, heir class or subclass
- The original class is known as base class, parent class or superclass

Object–oriented programming: heritage

How to indicate that a class inherits from another?

```
class subClassName (superClassName):
```

- If both are in the same file, the superclass must be defined before the subclass
- The subclass inherits all the methods from the superclass
- If a method is simultaneously defined in both classes, the one of the subclass is considered
- If the subclass has its own method `__init__`, this one must explicitely call to the base class `__init__` method

Object–oriented programming: heritage

```
class polygon:
  def __init__(self, np=0):
      self.numpoints = np
      self.__color = 'red'
  def get_color (self):
      return self.__color
  def set_color (self, c):
      self.__color = c
  def surface (self):
      pass
```

```
class triangle (polygon):
    def __init__(self):
      polygon.__init__(self,3)
      self.vertices = None
    def set_pts(self,pt0,pt1,pt2):
      if((pt0[0]==pt1[0]
          and pt0[0]==pt2[0])
       or (pt0[1]==pt1[1]
           and pt0[1]==pt2[1])):
        print ' Error'
        # print (' Error')
      else:
        self.vertices=[pt0,pt1,pt2]
    def surface (self):
      [x0,y0]=self.vertices[0]
      [x1,y1]=self.vertices[1]
      [x2,y2]=self.vertices[2]
      s = abs((x1-x0)*(y1-y0)-
          (x1-x0)*(y1-y0))/2.0
      return s
```

```
t1 = triangle ()
t1.set_pts ([0,0],[1,0],[1,1])
t2 = triangle ()
t2.set_pts ([-1,0],[0,0],[-1,2])
t2.set_color ('blue')
print 't2 surface: ', t2.surface ()
# print ('t2 surface: ', t2.surface ())
```

Object–oriented programming: heritage

We can also use a class defined in a different file:

In file `polygon.py`:

```
class polygon:
    ...
```

In file `triangle.py`:

```
from polygon import polygon
class triangle (polygon):
    ...
```

Exercise: Write the `rectangle` class, child of the `polygon` class

- Each rectangle is defined by an lower left and an upper right vertices
- Write the surface method
- Write a method that gives the four vertices of the rectangle
- From an external code, build different triangles and rectangles and test the implemented methods.

Object–oriented programming: further topics

- Any variable of the form `self.var` is public and accesible from any method of the class, and from the program where the class instance is used

- Writing `__` before a variable or a function, we create a *pseudo-private* variable or function: `self.__privatevar`

Object–oriented programming: further topics

- Attribute data (also called *instance variables* in JAVA and *member variables* in C++) are the variables associated to a specific instance of a class
  - PYTHON also has class attributes, which are variables associated to the class but not to a specific instance
- From outside the class, we access the attribute `nombre` of the instace `obj` by writing obj.name
- Inside the class, we do `self.name`
- An attribute exists since the moment a value is assigned to it

Instance attributes and class attributes

```
1  class Point:        # File: ej1_class_ptos.py
2      """ Class Point """
3
4      numPts = 0
5
6      def __init__ (self, x=0, y=0):
7          Point.numPts += 1   # variable of the class, shared
               by all its instances
8          self.x = x          # variable of the instance
9          self.y = y
10
11     def print_pt (self):
12         print ' Point: ( ', self.x,' , ', self.y,' )'
13        # print (' Point: ( ', self.x,' , ', self.y,' )')
14
15 if __name__ == '__main__':
16   pt1 = Point ()
17
18   pt2 = Point (3,5)
19
20   pto1.print_pt ()
21   print 'p2: (%f, %f)' % pto2.x, pto2.y
22   print 'Total number of points: ', pto1.numPts
23   # print ('p2: (%f, %f)' % pto2.x, pto2.y)
24   # print ('Total number of points: ', pto1.numPts)
```

Polymorphism

- **Polymorphism** is the capacity of objects and methods of a class to react in a different way depending on the parameters or arguments they receive

- Being a dynamical language, polymorphism is no very important in PYTHON

NumPy

Working with loops in Python is, in general, much slower than in compiled language (Fortran, C, C++)

However, the NumPy library allows vectorial operations in an efficient way, with a speed comparable to the compiled languages

```
>>> from numpy import *      # import all NumPy methods
>>> import numpy as np
```

NumPy allows, for example:

- addition, subtraction and matrix products, with no need of operating with its components
- matrices inversion and solving systems of linear equations
- automatically generation of special matrices (null matrix, identity, ...)
- eigenvalues and eigenvectors computing

NumPy

An *array* can be created from a *list*:

```
>>> a0 = [0, 1.2, 4, -9.1, 5]
>>> a1 = array (a0)
>>> b0 = [[-1, 1.], [2, 1]]
>>> b1 = array (b0)
```

It is also possible to create a *list* from an *array*:

```
>>> x.tolist ()
```

Some particular *arrays*:

```
>>> a1 = zeros (3[,dtype])      # dtype = 'f', 'd', 'i'
>>> a2 = zeros ([4,2])
>>> a3 = ones (2[,dtype])      # dtype = 'f', 'd', 'i'
>>> a4 = ones ([3,5])
>>> a5 = eye (3)
>>> a6 = linspace (a,b,n)      # by default, n = 50
>>> a7 = logspace (a,b,n)
```

NumPy

```
>>> u = np.array ([0., 1., 2., 3.])
>>> v = np.array ([-2., 4., -1., 6.])
>>> zu = np.zeros_like (u)
>>> zv = np.ones_like (v)
>>> u [u > v]
array ([ 0., 2.])
>>> v [u > v]
array ([-2., -1.])
```

NumPy

We can obtain and change the dimensions of an *array*:

```
>>> x.shape      (or   >>> shape (x))
>>> x.shape[0]
>>> x.shape[1]
>>> x.shape = (3,2)      >>> x = reshape (x,(2,3))
```

which is different to:

```
>>> x.size      (or   >>> size (x))
```

Other methods:

- `>>> diag (x)`: returns the main diagonal of the array `x`
- `>>> diag (v)`: builds a diagonal matrix with the components of vector `v`
- `>>> triu (x)`: returns the upper triangular part of `x`
- `>>> tril (x)`: returns the lower triangular part of `x`

```
>>> inverse (x)
>>> transpose (x)
```

NumPy

Access to array components:

```
>>> a[2,3]; a[:,0]; a[1,:]; a[0:2,1:4]
```

```
for i in range (a.shape [0]):
    for j in range (a.shape [1]):
        b[i,j] = 3 * a[i,j] - 1;
```

An equivalent, and more efficient, implementation is:

```
b = a;
multiply (b, 3, b);
subtract (b, 1, b);
```

or:

```
b = 3 * a - 1.0;
```

Array product

- Array product: `C=numpy.dot(A,B)`, where `A` and `B` are NumPy arrays of size $m \times n$ and $n \times p$

```
>>> import numpy as np
>>> A = np.array ([[-1,0,0],[0,-3,0],[0,0,1]])
>>> B = np.array ([[-1,2],[-3,0],[1,-3]])
>>> np.dot (A,B)
    array([[ 1, -2],
           [ 9, 0],
           [ 1, -3]])
```

  Remark: The product `A*v`, being `A` a matrix and `v` a vector, it's independent on whether `v` is a row or column vector

- Inner product: `numpy.inner(v,w)`, where `v` and `w` are one–dimension arrays

```
>>> v = np.array ([1, -2, 3])
>>> w = np.array ([-2, 3, 4])
>>> np.inner (v,w)
    4
```

Relational and logical operators

Relational operators: `<` , `>` , `<=` , `>=` , `==` , `!=`

- when applied to arrays of the same shape, they operate element to element and returns an array which components are `True` and/or `False`
- when applied to an array and a scaler value, they compare each component with the scalar, and returns an array which components are `True` and/or `False` values

Logical operators: `and` , `or` , `not`

- they can be combined with the relational operators to verify multiple conditions.

NumPy

Other methods:

- `max`, `min`

```
>>> a.max ()
>>> max (a)      # only if a is similar to a single list
```

- `sum`, `cumsum`
- `prod`, `cumprod`
- `mean`, `std`
- `sort`

```
>>> a = array ([1., 2.],[3, 4],[5, -3])
>>> sort (a)
array ([[ 1., 2.],
        [ 3., 4.],
        [-3., 5.]])
```

- `rot90`, `flipud`, `fliplr`

NumPy

The `Numpy` methods `sin`, `cos`, `tan`, `log`, `log10`, `exp`, `sqrt`, `arcsin`, `sinh`, `**` (power), ... act over each componente of an `array`.

It is possible to implement our own methods, where the function acts over each component of an `array`. For example, if we write a file `func01.py` with the following contents

```
def f (x):
    return x ** 2
```

then we can do:

```
>>> from numpy import array
>>> import func01
>>> x = array ([[-1, 0],[2, 3]])
>>> x2 = func01.f (x)
```

The following result is obtained:

```
array([[1, 0],
       [4, 9]])
```

NumPy

Let's implement the following methods:

```
def f03 (x):
  if x < 0:
    return 0;
  else:
    return sin (x);
```

```
def f04 (x):
  n = size (x)
  r = zeros (n,float)
  for i in range (n):
    if x[i] < 0:
      r[i] = 0.0;
    else:
      r[i] = sin (x[i]);
  return r
```

```
def f05 (x):
  ind = less (x,0);
  r = sin (x)
  r = where (ind, 0.0, r)
  return r;
```

NumPy

Let's write:

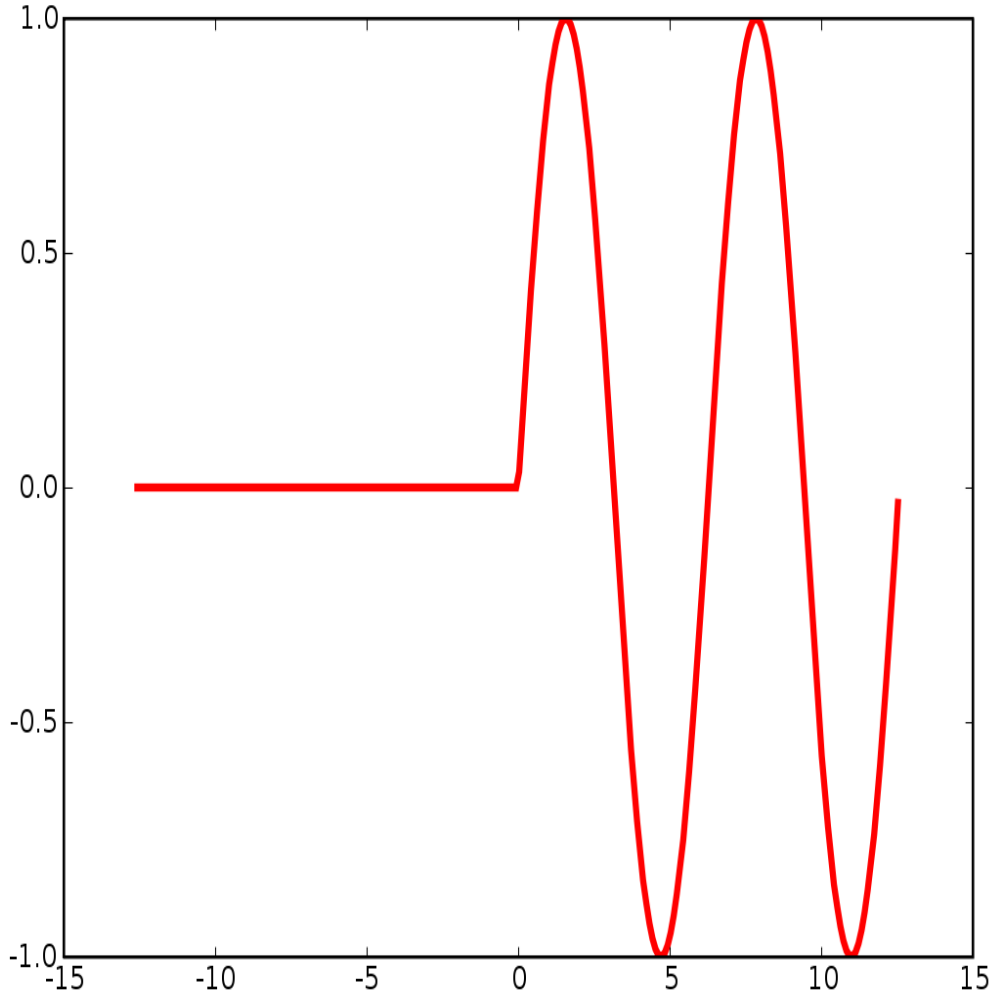
```
import numpy as np
from pylab import plot, show

x = np.arange
(-4*np.pi,4*np.pi,0.01)

y03 = f03 (x)
y04 = f04 (x)
y05 = f05 (x)

p = plot (x,y04,'r-')
show ()
```

The call to function `f03` will return the following error:
`ValueError: the truth value of an array with more than one element is ambiguous`

NumPy

Let's write:

```
import numpy as np
from pylab import plot, show

x = np.arange
(-4*np.pi,4*np.pi,0.01)

y03 = f03 (x)
y04 = f04 (x)
y05 = f05 (x)

p = plot (x,y04,'r-')
show ()
```



The call to function `f03` will return the following error:

```
ValueError:  the truth value of an array with more than
one element is ambiguous
```

`all` and `any` methods

If `x` is a one–dimensional *array*,

`any (x)`
- `= True`, if any array element of `x` is nonzero
- `= False`, if all elements of `x` are zero

`all (x)`
- `= True`, if all elements of `x` are nonzero
- `= False`, if `x` has at least a null element

If `a` is a multidimensional *array*:

`numpy.any (a)`, `a.any ()`, `numpy.all (a)`, `a.all ()`

Loops vectorization using NumPy

When possible, the loops must be vectorized:

```
import time
from numpy import *
n = input ('Introduce N: ')

c1 = time.clock ()
x = zeros (n)
for i in range (n):
    x[i] = 3.0 - 2*i + i*i
c2 = time.clock ()

ind = (linspace (0,n-1,n))
y = 3.0 - 2.*ind + ind*ind
c3 = time.clock ()

print 'Elapsed time:'
print '   (a) ', c2-c1, ' s'
print '   (b) ', c3-c2, ' s'
```

The `Matrix` object of NumPy

- NumPy provides a special object to handle matrices (bidimensional **arrays**): `matrix`. It is a subclass of the `array` class. This object is used to make linear algebra operations
- The object `array` has a general purpose to handle multidimensional arrays. However, the object `matrix` simplifies algebraic operations

```
>>> from numpy import *
>>> A = matrix ('1 3 4; 5 6 9; 0, 1, 2')
           # or:  A = mat ('1 3 4; 5 6 9; 0, 1, 2')
>>> B = matrix ('-1 0 0; 0 -3 0; 0, 0, -4')
>>> A * B  # matrix multiplication
>>> A.T    # transpose
>>> A.I    # inverse matrix
```

The Matrix object of NumPy

- numpy.asarray (a, dtype=None, order=None): returns a as an array.

```
>>> from numpy import *
>>> m = matrix ('1 2; 5 8')
>>> a = asarray (m)   # a is a copy of m by reference
>>> a
array([[1, 2],
[5, 8]])
>>> m[0,0] = -99
>>> m
matrix ([[-99,   2],
[  5,   8]])
>>> a                 # if a changes, also m changes, and viceversa
array ([[-99,   2],
[  5,   8]])
>>> del a    # althoug a is deleted, m not
>>> m
matrix ([[-99,   2],
        [  5,   8]])
```

- numpy.asmatrix (data, dtype=None): transforms data in an object matrix

Polynomials handling in NumPy

- `Numpy` provides several methods to work with polynomials
- In particular, the class `poly1d` creates a `polynomial`–type object
- This class allows to build a polynimial using:
  - a `list` that specifies the coeffients, or
  - a `list` that contains the roots of the polynomial
- It is possible to work with `polynomial` objects in algebraic expressions. A `polynomial` can be derivated, integrated and evaluated. The `polynomial` objects can be added, subtracted, multiplied and divided.

Polynomials handling in NumPy

```
>>> import numpy as np
        # polynomial created from its coefficients
>>> p = np.poly1d ([3, 2, 0, 1])
>>> print (p)
   3     2
3 x + 2 x + 1

>>> p.r            # roots of the polynomial
array([-1.00000000+0.j        ,  0.16666667+0.5527708j ,
        0.16666667-0.5527708j])

>>> p(0.5)              # it evaluates the polynomial in 0.5.
                          # Analogous to np.polyval(p,0.5)
1.875
>>> p([2,-1,-10])     # evaluates the polynomial in 2,-1,10
array([   33,     0, -2799])

>>> print (p.deriv (m=2)) # prints second derivative of p
18 x + 4

>>> print (p.integ ())     # prints the primitive of p
      4          3
0.75 x + 0.6667 x + 1 x
```

Polynomials handling in NumPy

```
>>> c = np.poly ([2,-2]) # returns coefficients of the
                         # polynomial which roots are 2 and -2

>>> q = np.poly1d (c)
>>> print p + q           # sum of polynomials;
                          # equal to np.polyadd (p,q)
   3     2
3 x + 3 x - 3

>>> print p*q             # product of polynomials
   5     4       3     2
3 x + 2 x - 12 x - 7 x - 4

>>> q / p                 # polynomial division;
                          # equal to np.polydiv (q,p)
(poly1d ([0.]), poly1d ([1, 0, -4]))
```

Introduction to Python

Variables and data types

Python programming

Object–oriented programming

NumPy: Numerical Python

SciPy: Scientific Python

- Linear algebra
- Interpolation
- Numerical integration
- Optimization
- Some financial functions

Bidimensional graphics with MatPlotLib

Pandas

SciPy

What is SciPy?

- It is a Python module for scientific computation
- Python–type licence
- Developed by Enthought; its (main) authors are Eric Jones, Travis Oliphant, Pearu Peterson and Prabhu Ramachandran
- NumPy–based core
- It provides many routines for statistics, optimization, signal and image processing, ...

It is not included in the basic Python distribution, so it must be expressely installed

SciPy modules

SciPy includes several modules under the `scipy` *namespace*

- `Tools`:
  - `cluster`: vector quantization / kmeans
  - `fftpack`: Fourier transformed
  - `integrate`: numerical quadrature
  - `interpolate`: interpolation and ODE solvers
  - `linalg`: linear algebra routines
  - `misc`: several utilities (including the *Python Imaging Library*)
  - `ndimage`: n-dimensional images tools
  - `optimize`: optimizatin and root finding tools
  - `signal`: signal processing tools
  - `sparse`: *sparse* matrices
  - `stats`: statistical functions
- Other packages:
  - `io`: data input/output
  - `lib`: *wrappers* for external libraries (`BLAS, LAPACK`)
  - `special`: mathematical functions definiciones de muchas funciones usuales de matemáticas
  - `weave`: C/C++ interaction

SciPy `linalg` module

It is a linear algebra module. It is also available from `numpy`, but in a reduced version

```
from numpy import array
import scipy
from scipy import linalg
A = array ([[1,0,0,0],[0,-1,-1,0],[0,0,-1.,0],[0,0,0.,2]])
b = array ([2,3,4,0])
```

`linalg.det (A)`: `A` determinant

`linalg.inv (A)`: computes the inverse of `A`

`linalg.solve (A,b)`: solves the $Ax = b$ system

`linal.eig (A)`: computes eigenvalues and eigenvectors of `A`

`linal.eigvals (A)`: computes eigenvalues of `A`

SciPy `linalg` module

```
import numpy
from numpy import array
import scipy
from scipy import linalg

A = array ([[1,0,0,0],[0,-1,-1,0],[0,0,-1.,0],[0,0,0.,2]])
b = array ([2,3,4,0])

detA = linalg.det (A)
iA = linalg.inv (A)
c = linalg.solve (A,b)
[eigvals,eigvecs] = linalg.eig (A)
aval = linalg.eigvals (A)

aval2 = numpy.linalg.eigvals (A)
```

SciPy `linalg` module: linear systems

LU **factorization**:

Given a matrix A such that $\det(A_k) \neq 0$ $(k = 1, 2, ...)$, there exist a lower triangular matrix L and an upper triangular U such that: $A = LU$

```
A = array ([[60,30,20],[30,20,15],[20,15,12]])
linalg.lu (A,[permute_l=0, overwrite_a=0])
```

If `permute_l=0`, `p,l,u=linalg.lu(A)` $\Rightarrow$ `a = p*l* u`

If `permute_l=1`, `m,u=linalg.lu(A,permute_l=1)` $\Rightarrow$ `a = m*u`

Cholesky (LL^T) **factorization**:

Given a symmetric and defined positive matrix A, there exists a lower (upper) triangular matrix L (U) such that $A = LL^T$ $(A = U^T U)$

```
M = array ([[0.9, 0.06, -0.39, -0.24], \
   [0.06, 1.604, 0.134, 0.464], \
   [ -0.39, 0.134, 2.685, 0.802], \
   [ -0.24,0.464, 0.802, 1.977]])
linalg.cholesky (M)
```

SciPy `linalg` module: linear systems

QR factorization:

Given an inversible matrix A, there exist an orthogonal matrix Q and an upper triangular matrix R such that:

$$A = QR$$

```
A = array ([[1,-2,1],[-1,3,2],[1,-1,-4]])
linalg.qr (A)
```

SciPy `interpolate` module

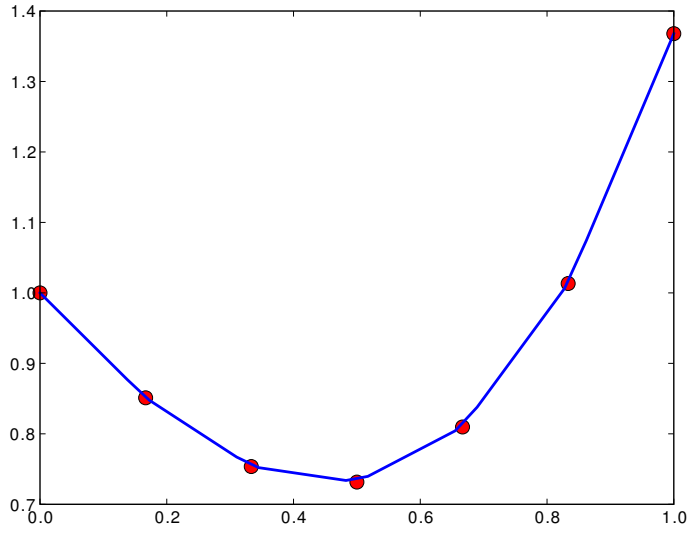
`scipy.interpolate` package provides two general kinds of interpolation tools:

- Linear unidimensional interpolation
- 1–D and 2–D cubic splines (based in `FITPACK` library, written in `Fortran`)

SciPy interpolate module: 1–D interpolation

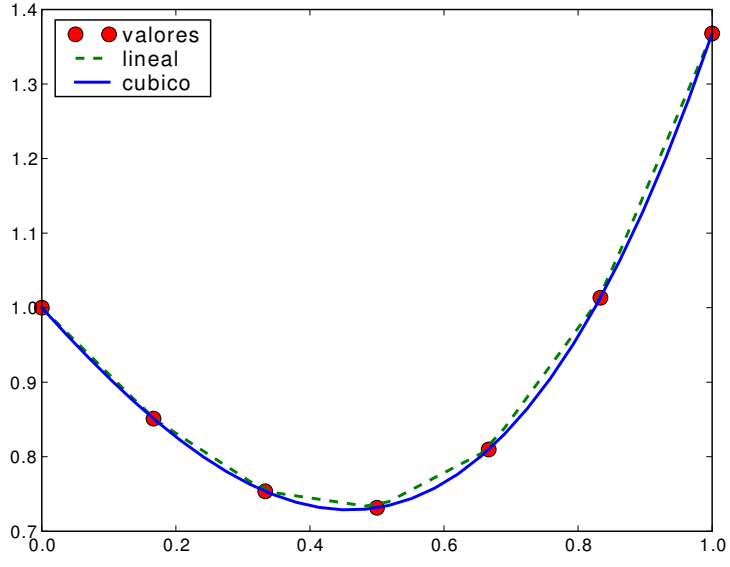
Assume we want to represent the function $f(x) = e^{-x} + x^3$ in the interval $[0, 1]$, but we only know the value of f in seven points:

```
1  from scipy import interpolate as inter
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  x = np.linspace (0., 1., 7)
6  y = np.exp (-x) + x**3
7
8  f = inter.interp1d (x, y)
9
10 xa = np.linspace (0,1,30)
11 ya = f(xa)
12         # evaluates f in
13         # every point of xa
14
15 for i in range (0, xa.size):
16     print xa[i], '\t', ya[i]
```



SciPy interpolate module

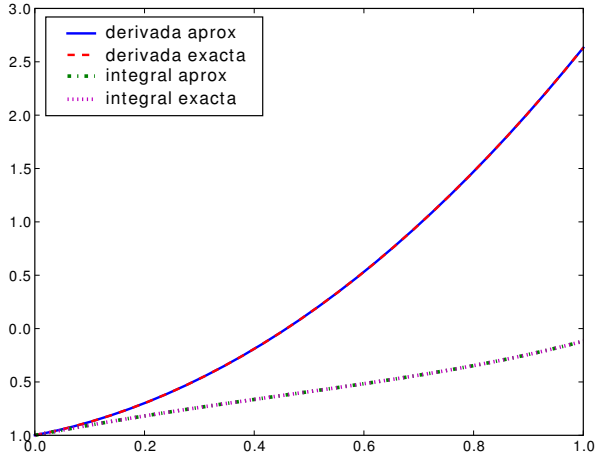
```
1  from scipy import interpolate as inter
2  import numpy as np
3  import pylab
4
5  x = np.linspace (0.,1., 7)
6  y = np.exp(-x) + x**3
7
8  s1_f = inter.splrep (x,y,k=1)
9         # linear spline
10 s3_f = inter.splrep (x,y)
11        # cubic spline
12
13
14 xs = np.linspace (0,1,30)
15
16 # Evaluation of the spline in xs points
17 y_s1 = inter.splev (xs,s1_f,der=0)
18
19 # der: derivative order
20
21 y_s3 = inter.splev (xs,s3_f,der=0)
```



SciPy interpolate module

It is also possible to evaluate the derivative and the integral of a spline

```
# First derivative (der = 1)
dys = inter.splev (xs,sc_f,der=1)

# Exact derivative value
deys = -np.exp(-xs)+3.0*xs**2
print 'Error Derivada: ',
      max(np.abs(deys-dys))

# Integral
iys = np.zeros_like (xs)
for i in range (0,xs.size):
    iys[i] = inter.splint (0.0,xs[i],sc_f)
iys += -1

ieys = -np.exp(-xs) + 1.0/4.0*xs**4
print 'Integral error: ',  \
      max (np.abs(iys-ieys))
```



SciPy `integrate` module

The `scipy.integrate` module allows:

- numerically approximate the integral of a function in a (not necessarily finite) interval
  - `quadrature (f, a, b, args=(), tol=1.5e-8, maxiter=50)`: integrates function `f` between `a` and `b` by a quadrature formula with tolerance `tol`.
  - `quad (f, a, b, ...)`: idem, using QUADPACK library
  - `romberg (f, a, b, tol=1.48e-8, show=0, divmax=10)`: Romberg integration
  - `dblquad (f, a, b, gfun, hfun, args=(), epsabs=1.5e-8, epsrel=1.5e-8)`: 2–D integral
  - `tplquad (f, a, b, gfun, hfun, qfun, rfun, args=(), epsabs=1.5e-8, epsrel=1.5e-8)`: 3–D integral
- numerically approximate the solution of systems of ODEs
  - `odeint`

SciPy integrate module

Example:

$$\int_0^{2\pi} cos^2(x)dx$$

```
>>> from scipy import integrate
>>> import math
>>> def f(x):
...     return math.cos(x)**2
...
>>> val,error = integrate.quad (f,0.0,2.0*math.pi)
>>> val, error
    (3.1415926535897931, 2.3058791641795214e-09)
```

SciPy integrate module

```
>>> from scipy import integrate
>>> import scipy
>>> import math
>>> def f(x):
...     return math.sqrt(x+1)
...
>>> def f2(x):
...     return 1.0/(x**2+1)
...
>>> for function in [f,f2]:
...  val,error=integrate.quad(function,0.0,scipy.Inf)
...  print 'Integral ',funcion.__name__,': value= ',val, \
     'error= ',error
Warning: The integral is probably divergent, or slowly
    convergent
Integral f: value= -0.666666666676 error= 3.01820790582e-11
Integral f2: value= 1.57079632679 error= 2.5777919189e-10
```

SciPy `integrate` module

- Numerical solution of a first order ODE system:

$$\frac{d\vec{y}}{dt} = f(\vec{y}, t)$$

- Pendulus equation (second order):

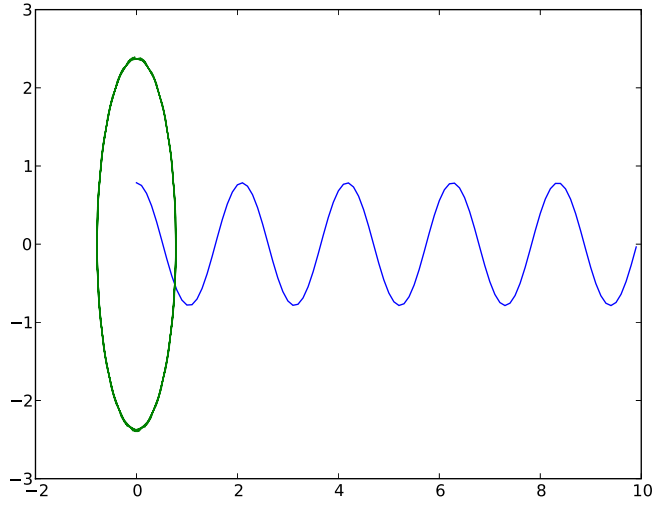
$$\frac{d^2\theta}{dt^2} = -\frac{g}{L} sin\theta$$

It is equivalent to a first order system:

$$\begin{cases} \frac{d\theta}{dt} = v \\ \frac{dv}{dt} = -\frac{g}{L} sin(\theta) \end{cases}$$

SciPy integrate module

```
1  import scipy, scipy.integrate, pylab
2
3  def dydt (y,t,g,L):
4      theta, v = y
5      return [v, -(g/L)*scipy.sin(theta)]
6
7  g = 9.8
8  L = 1.0
9
10 times = scipy.arange (0.,10., 0.1)
11 y0 = [scipy.pi/4.0,0.]    # initial cond
       .
12 y_trajectory = scipy.integrate.odeint \
13         (dydt,y0,times,args=(g,L))
14    #args: optional arguments
15
16 pylab.plot (times, y_trajectory[:,0])
17 pylab.plot (y_trajectory[:,0], \
18               y_trajectory[:,1])
19 pylab.show ()
```



SciPy `optimize` package

The `scipy.optimize` package provides different functionalities:

- Optimization methods:
  - Nelder–Mead simplex algorithm: `fmin`
  - BFGS algorithm: `fmin_bfgs`
  - Newton / Conjugate Gradient methods: `fmin_ncg`
  - Least squares: `leastsq`
  - Constrained least squares: `fmin_slsqp`
- Minimizing a scalar function: `brent`, `fminbound`
- Search for non linear equations roots: `fsolve`

SciPy: roots of nonlinear equations

- Roots of a polynomial equation: `roots`

Example: $x^2 - 5x + 6 = 0$

```
>>> import scipy
>>> scipy.roots([1, -5, 6])
array([ 3.,  2.])
```

Example: $-x^3 + 2x^2 - 8x - 1 = 0$

```
>>> import scipy
>>> scipy.roots([-1,2,-8,-1])
array([ 1.06055549+2.67060141j,  1.06055549-2.67060141j,
       -0.12111098+0.j        ])
```

SciPy: roots of nonlinear equations

- Roots of a general nonlinear equation: `fsolve`.
  It is a wrapper to `MINPACK` `hybrd` and `hybrj` algorithms.

Example: $x + 2\cos(x) = 0$

```
>>> from math import cos
>>> def func (x):
...     return x + 2*cos(x)
>>> from scipy.optimize import fsolve
>>> x0 = fsolve (func,0.3)
>>> print x0
-1.02986652932
```

Example: $x_0 \cos(x_1) = 4;\ x_0 x_1 - x_1 = 5$

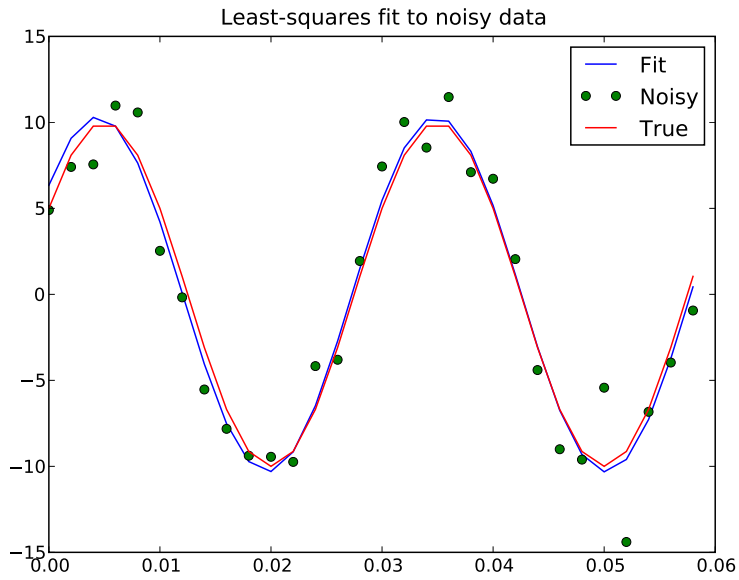
```
>>> from math import cos
>>> def func (x):
...     return [x[0]*cos(x[1]) -4.0, x[0]*x[1]-x[1] -5.0]
>>> from scipy.optimize import fsolve
>>> r = fsolve (func, [1., 1.])
>>> print r
array([ 6.50409711,  0.90841421])
```

SciPy: least squares

Goal: given the couples (x_i, y_i), we search the function that best fits the data according the least square error

For example, we search the parameters A, k and θ in order to have the best approximation to: $y = A\sin(2\pi kx + \theta)$

```
from numpy import *
x = arange(0,6e-2,6e-2/30)
A,k,theta = 10, 1.0/3e-2, pi/6
y_true = A*sin(2*pi*k*x+theta)
y_meas = y_true + 2*random.randn(len(x))

def residuals(p, y, x):
    A,k,theta = p
    err = y - A * sin (2*pi*k*x+theta)
    return err
def peval(x, p):
    return p[0] * sin (2*pi*p[1]*x+p[2])
p0 = [8, 1/2.3e-2, pi/3]
print array (p0)

from scipy.optimize import leastsq
plsq = leastsq (residuals, p0, args=(y_meas, x))
print plsq[0]
print array ([A, k, theta])

import matplotlib.pyplot as plt
plt.plot (x,peval(x,plsq[0]),x,y_meas,'o',x,y_true)
plt.title ('Least-squares fit to noisy data')
plt.legend (['Fit', 'Noisy', 'True'])
plt.show ()
```



NumPy financial functions

- `fv`: future value
- `pv`: present value
- `npv`: net present value
- `pmt`: payment against loan principal plus interest
- `ppmt`: payment against loan principal
- `ipmt`: interest portion of a payment
- `irr`: internal rate of return
- `mirr`: modified internal rate of return
- `nper`: number of periodic payments
- `rate`: rate of interest per period

The plot command

```
from pylab import *
plot (x,y,properties)
```

- `x` and `y` are real arrays of the same dimension
- `plot (x,y)`: plots `x` in abscissa axis and `y` in the vertical axis

Property	Abbrev.	Description
`alpha`	`-`	Float value in (0, 1), from transparent to opaque
`antialiased`	`aa`	True or False; uses antialiasing rendering
`color`	`c`	Matplotlib color
`label`	`-`	Optional string used in the legend
`linestyle`	`ls`	Line style
`linewidth`	`lw`	Float, linewidth
`marker`	`-`	A valid marker style
`markeredgewidth`	`mew`	Linewidth around the marker
`markeredgecolor`	`mec`	Marker border color
`markerfacecolor`	`mfc`	Marker color
`markersize`	`ms`	Markersize

The `plot` command

Example:

```
x = numpy.arange (-10,10.5,0.5)
y = x ** 2
```

- By default plotting: solid blue line without markers

  ```
  plot (x,y)
  ```

- Specifying properties:

  ```
  plot (x,y, c='r', ls='--', marker='s', ms=4, mfc='g')
  plot (x,y, 'r--s', ms=4, mfc='g')
  ```

  plots a dashed red line, with square green markers and black border

- Showing the plot in a GTK window:

  ```
  show()
  ```

The `plot` command

Example:

```
x = numpy.arange (-10,10.5,0.5)
y = x ** 2
```

- By default plotting: solid blue line without markers

  ```
  plot (x,y)
  ```

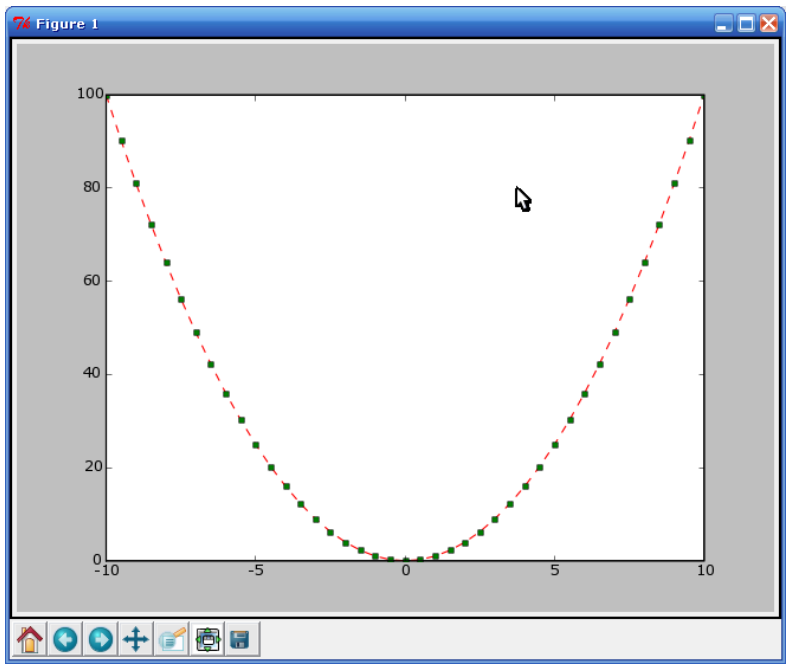
- Specifying properties:

  ```
  plot (x,y, c='r', ls='--', marker='s', ms=4, mfc='g')
  plot (x,y, 'r--s', ms=4, mfc='g')
  ```

  plots a dashed red line, with square green markers and black border

- Showing the plot in a GTK window:

  ```
  show()
  ```



The `plot` command

Symbol	Color	Symbol	Description
`y`	yellow	`.`	points
`m`	magenta	`o`	circles
`c`	cyan	`x`	$\times$ marker
`r`	red	`+`	$+$ marker
`g`	green	`s`	square marker
`b`	blue	`d`	diamond marker
`w`	white	`^`	triangle up marker
`k`	black	`v`	triangle down marker
`0.75`	grayscale $[0, 1]$	`>`	triangle right marker
`#636fb0`	RGB (hexadecimal)	`<`	triangle left marker
`(0.9,0.111,1)`	RGB (tuple)	`p`	pentagon marker
		`h`	star marker

Symbol	Line style
`-`	continuous line
`:`	dotted line
`-.`	dash-dot line
`--`	dashed line

All symbols are given as string, except the RGB tuple

Graphics control

- `hold (value)`, `value=True/False`: Sets the 'hold' state. If `True`, subsequent plot commands will be added to the current axes. If `False`, the current axes and figure will be cleared on the next plot command. Default value is `True`
- `grid (value)`, `value=True/False`: Turns the axes grids on or off. Default value is `False`
- `box (value)`, `value=True/False`: Turns the axes box on or off. Default value is `False`

- `title ('string')`: Sets a title of the current axes
- `xlabel ('string')`: Sets the X axis label of the current axis; it can be eliminated by writing `xlabel off`
- `ylabel ('string')`: Sets the Y axis label of the current axis

Graphics control

- `text (x,y,s (,parameters))`: adds text in string `s` to axis at location (x, y), data coordinates. The value of the parameters are:

```
horizontalalignment = 'left'/'right'/'center'
verticalalignment = 'top'/'bottom'/'center'
rotation = grades
multialignment = 'left'/'right'/'center'
```

- `legend (plots-list, labels-list, loc=value, shadow=False/True)`: Places a legend on the axes. To make a legend for lines which already exist on the axes (via plot for instance), simply call this function with an iterable of strings, one for each legend item.
  `loc` specifies the location of the legend.

```
loc = 'best','center','upper','right','left','lower',
    'upper right','center right','lower right'
```

  If `loc` is not especified, the legend is placed in the best possible place

Graphics control

- `idf = figure (num)`: Creates a new figure. If `num` is not provided, a new figure will be created, and the figure number will be incremented. The figure objects holds this number in a number attribute. If `num` is provided, and a figure with this id already exists, makes it active, and returns a reference to it. If this figure does not exists, creates it and returns it. If `num` is a string, the window title will be set to this figures `num`.
- Functions associated to `figure`, that can be used from the `ipython` console:
  - `close (idfig)` or `close (num)`: closes the figure asociated to the reference `idf` or the number `num`
  - `clf()`: clears the active figure

```
>>> from pylab import *
>>> f1 = figure (1)
>>> x = arange (-10,10,0.5)
>>> y = x**2
>>> z = sqrt(abs(x))
>>> plot (x,y)              # Plot in f1
>>> f2 = figure(2)
>>> plot (x,y,'g:')         # Plot in f2
>>> figure (1)
>>> plot (x,10*z)           # Plot in f1
>>> close (f1)
```

Axis control

`v = axis ([xmin, xmax, ymin, ymax])`: Convenience method to get or set axis properties. Sets the maximum and minimum values on each axis. Returns `v` as the list `[xmin, xmax, ymin, ymax]`

`axis('equal')`: changes limits of x or y axis so that equal increments of x and y have the same length

`axis('scaled')`: achieves the same result by changing the dimensions of the plot box instead of the axis data limits

`axis('image')`: is `'scaled'` with the axis limits equal to the data limits

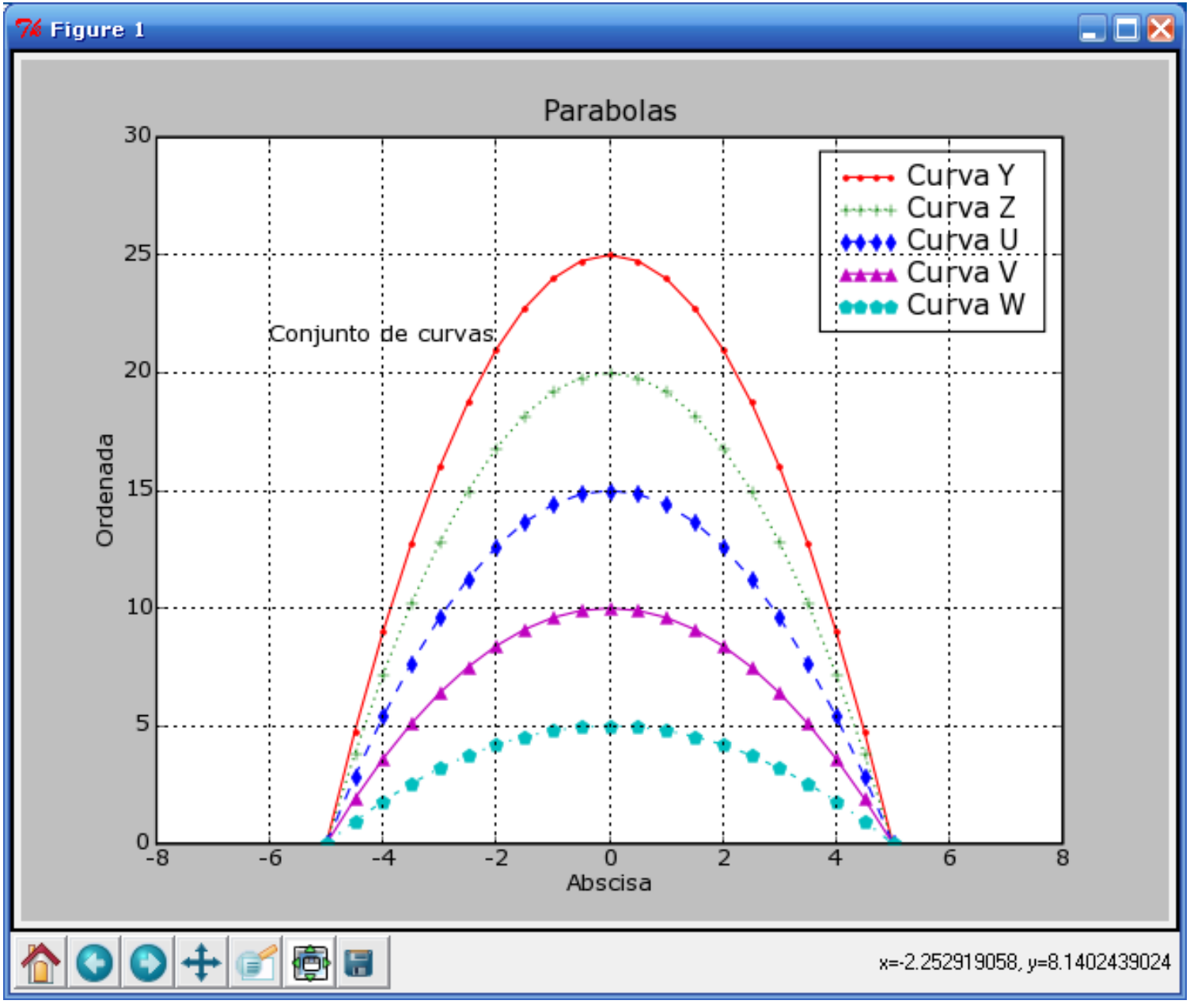
`axis('off')`: turns off the axis lines and labels

`axis('on')`: turns on the axis lines and labels

Example ex1_plot2d.py

```
from pylab import *
x = arange (-5,5+0.5,0.5)
y = - x*x + 25
plot (x,y,'r-', marker='.',  label='Y Curve')
grid (True)
z = -0.8*x*x+20
plot (x,z,'g:', marker='+',  label='Z Curve')
u = -0.6*x*x+15
plot (x,u,'b--', marker='d',  label='U Curve')
v = - 0.4*x*x + 10
plot (x,v,'m-', marker='^',  label='V Curve')
w = - 0.2*x*x + 5
plot (x,w,'c-.', marker='p',  label='W Curve')
axis ([-8 ,8 ,0 ,30])
legend (loc='best')
text (-6, 21, 'Curves set')
xlabel ('Abscissa'); ylabel ('Ordinate'); title ('Paraboles')
savefig ('simpleplot.png')
show ()
```

Example `ex1_plot2d.py`



The `subplot` and `axes` commands

They both add axes to the figure

- `subplot(numRows,numCols,plotnum)`: Returns a subplot axes positioned by the given grid definition. Where **`numRows`** and **`numCols`** are used to notionally split the figure into `numRows` $\times$ `numCols` sub-axes, and `plotnum` (starts at 1) is used to identify the particular subplot.
  - If **`numRows`**, **`numCols`** and **`plotnum`** are all less than 10, commas are not necessary: `subplot(numRowsnumColsplotnum)`

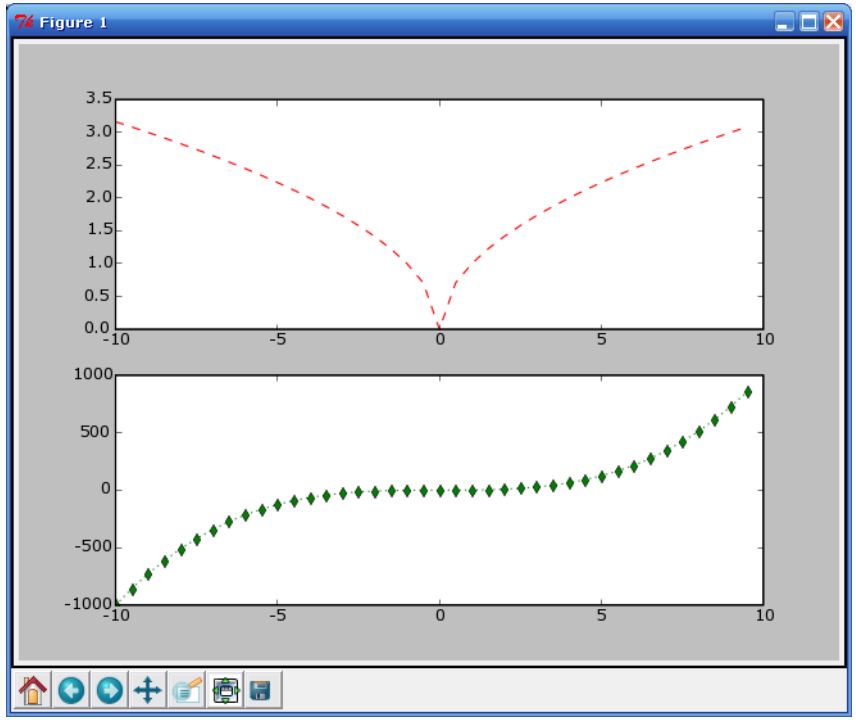
```
f = figure (1)
x = arange (-10,10,0.5)

subplot (211)
   # subplot (2,1,1)
plot(x,sqrt(abs(x)),'r--')

subplot (212)
plot (x,x**3,'g:d')
show ()
```

The subplot and axes commands

They both add axes to the figure

- subplot(numRows,numCols,plotnum): Returns a subplot axes positioned by the given grid definition. Where **numRows** and **numCols** are used to notionally split the figure into numRows × numCols sub-axes, and plotnum (starts at 1) is used to identify the particular subplot.
  - If **numRows**, **numCols** and **plotnum** are all less than 10, commas are not necessary: subplot(numRowsnumColsplotnum)

```
f = figure (1)
x = arange (-10,10,0.5)

subplot (211)
   # subplot (2,1,1)
plot(x,sqrt(abs(x)),'r--')

subplot (212)
plot (x,x**3,'g:d')
show ()
```



The subplot and axes commands

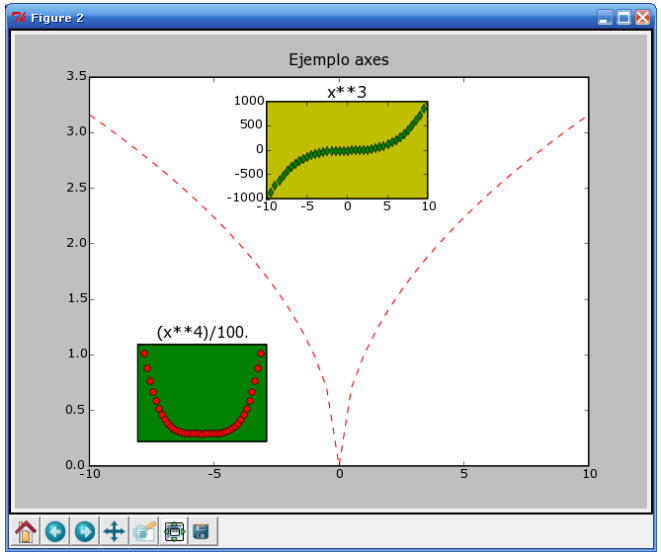
- axes(rect,axisbg='color'): Adds an axis to the figure. The axis is added at position rect specified by:
  - rect=[left,bottom,width,height] and measured in normalized units $(0, 1)$ respect to figure
  - axisbg is the background color, that it is white by default

```
f = figure (2)
x = arange (-10,10.5,0.5)

# Main figure
p1 = plot (x, sqrt(abs(x)),'r--')
title ('Ejemplo axes')

a1 = axes ([.4, .65, .25, .2],
    axisbg='y')
plot (x,x**3,'g:d')
title ('x**3')

a2 = axes ([.2, .15, .2, .2],
    axisbg='g', xticks=[],  yticks=[])
plot (x,(x**4)/100.,'r-.o')
title ('(x**4)/100.')
axis ([-11,11,-10,110]);     show ()
```

The subplot and axes commands

- `axes(rect,axisbg='color')`: Adds an axis to the figure. The axis is added at position `rect` specified by:
  - `rect=[left,bottom,width,height]` and measured in normalized units $(0, 1)$ respect to figure
  - `axisbg` is the background color, that it is white by default

```
f = figure (2)
x = arange (-10,10.5,0.5)

# Main figure
p1 = plot (x, sqrt(abs(x)),'r--')
title ('Ejemplo axes')

a1 = axes ([.4, .65, .25, .2],
    axisbg='y')
plot (x,x**3,'g:d')
title ('x**3')

a2 = axes ([.2, .15, .2, .2],
    axisbg='g', xticks=[],  yticks=[])
plot (x,(x**4)/100.,'r-.o')
title ('(x**4)/100.')
axis ([-11,11,-10,110]);     show ()
```



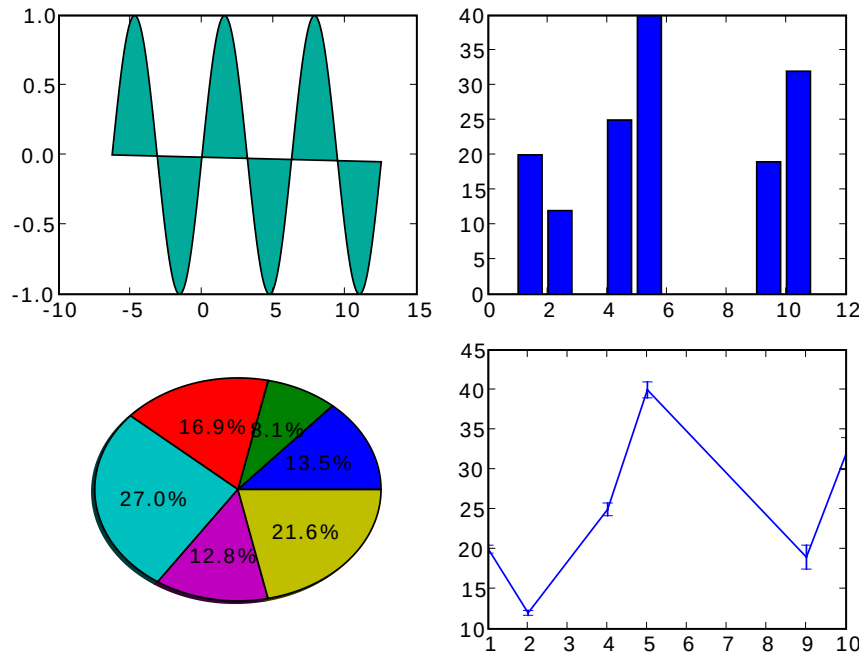
Other commands

- `fill (x,y,c)`: Plots filled polygons which vertices are the components $\{x, y\}$, and it's filled by the color `c` (the color can be specified by a string or in hexadecimal format `'#aaff01'`)
- `bar`: makes a bar plot
- `barh`: makes a horizontal bar plot
- `pie`: plots a pie chart
- `errorbar`: plots an errorbar graph

- `savefig ('namefile',dpi=150)`: saves the figure in the file `namefile`
  - If the extension is not specified, it saves the figure in `.png`. It is possible to save the figure in `.eps`, `.svg`
  - `dpi`: the resolution in dots per inch
- `draw ()`: redraws the current figure

Example: ex4_plot2d.py

```
from pylab import*

subplot (2,2,1)
x1 = arange (-4*(pi/2.),8*(pi/2.),0.05)
y2 = sin (x1)
fill (x1,y2,'#01aa99')
subplot (2,2,2)
x = [1 ,2, 4, 5 ,9, 10]
y = [20 ,12, 25, 40, 19, 32]
bar (x,y)
subplot (2,2,3)
fracs = [20,12,25, 40,19,32]
pie (y,autopct='%1.1f%%', shadow=True)
subplot (2,2,4)
e = [0.5, 0.3 ,0.8, 1.0, 1.5, 2.0]
errorbar (x,y,e)

savefig ('grafs.eps')
show ()
```



The setp and getp commands

- setp(obj,property): Sets a property on an artist object.

```
>>> line1 = plot (x,y)
>>> setp (line1, linestyle = '--', color = 'r')
```

setp(obj): allows to see all the properties, of the object obj, that can be set, and their possible values.

```
>>> setp (line1)
```

obj.set_someproperty(value): Sets the value of the property named someproperty:

```
>>> f1 = figure(1)
>>> f1.set_figwidth(5)
```

- getp(obj): gets object properties of (obj)

```
>>> l1 = plot (x,y)
>>> getp (l1)
```

obj.get_someproperty(): returns the value of the property called someproperty:

```
>>> l1 = plot (x,y)
>>> l1[0].get_xdata()
```

Animations

To visualized graphic animations with matplotlib, we can use:

- ion(): turns interactive mode on
- the class matplotlib.animation
- an asociated GUI

Using ion(): If the size of the used *arrays* is the same along all animation, we can use obj.set_xdata (objarray) and obj.set_ydata (objarray)

```
from pylab import *
import time

ion ()
step = 0.1
first = 0
x = arange(0, 2*pi+step,
step)
```

```
for n in range (0,10):
    y = ((-1)**n)*cos(x+n/10.)
    if first == 0:
        line=plot(x,y,'b--', marker='d')
        first = 1
    else:
        time.sleep (0.1)
        line[0].set_ydata (y)
        draw()      # Redraw
```

When the graphics animation finishs, the windows is automatically closed

Animations

Another posibility is to redraw the graphic at each time:

```
from pylab import *


ion ()
f1 = figure (figsize=(10,10))
ax = subplot (111)
x = arange (-3., 3.+0.2, 0.2)
y = arange (-3., 3.+0.2, 0.2)
[X,Y] = meshgrid (x,y)
for n in range (1,20):
    Z = sin(X) * cos(Y/(15.*n)) * cos(n*X*Y)
    # cla ()
    contour (X,Y,Z)
    draw ()     # Redraw
```

Animations

Using the class matplotlib.animation

Examples: http://matplotlib.org/examples/animation/index.html

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# First set up the figure, the axis, and the plot element we want to animate
fig = plt.figure()
ax = plt.axes(xlim=(0, 2), ylim=(-2, 2))
x = np.arange(0, 2*np.pi, 0.01)
line, = ax.plot(x, np.sin(x))
# animation function.  This is called sequentially
def animate(i):
    line.set_ydata(np.sin(x + i/10.0))  # update the data
    return line,

# Init only required for blitting to give a clean slate.
def init():
    line.set_ydata(np.ma.array(x, mask=True))
    return line,

# call the animator.  blit=True means only re-draw the parts that have changed.
ani = animation.FuncAnimation(fig, animate, np.arange(1, 200), init_func=init,
                              interval=55, blit=True)
plt.show()
```

Animations

- `init()` is the function which will be called to create the base frame upon which the animation takes place.
  It is important that this function return the line object, because this tells the animator which objects on the plot to update after each frame
- `animate(i)` is the animation function. It takes a single parameter, the frame number `i`.
  It returns a tuple of the plot objects which have been modified. This tells the animation framework what parts of the plot should be animated

Exercise

Consider $t \in [0, 2\pi]$ and the functions x and y given by:

$$x(t) = \sin(5t) \qquad\qquad y(t) = \cos(t)\,.$$

Write a code that implements the following:

1. Plot functions x and y
2. Plot function $y(x)$ (parametric equations)
3. Write a title introduced by the user
4. A natural number n given, plot (with four line styles) the curves containing the pairs $\{x, y\}$, with:

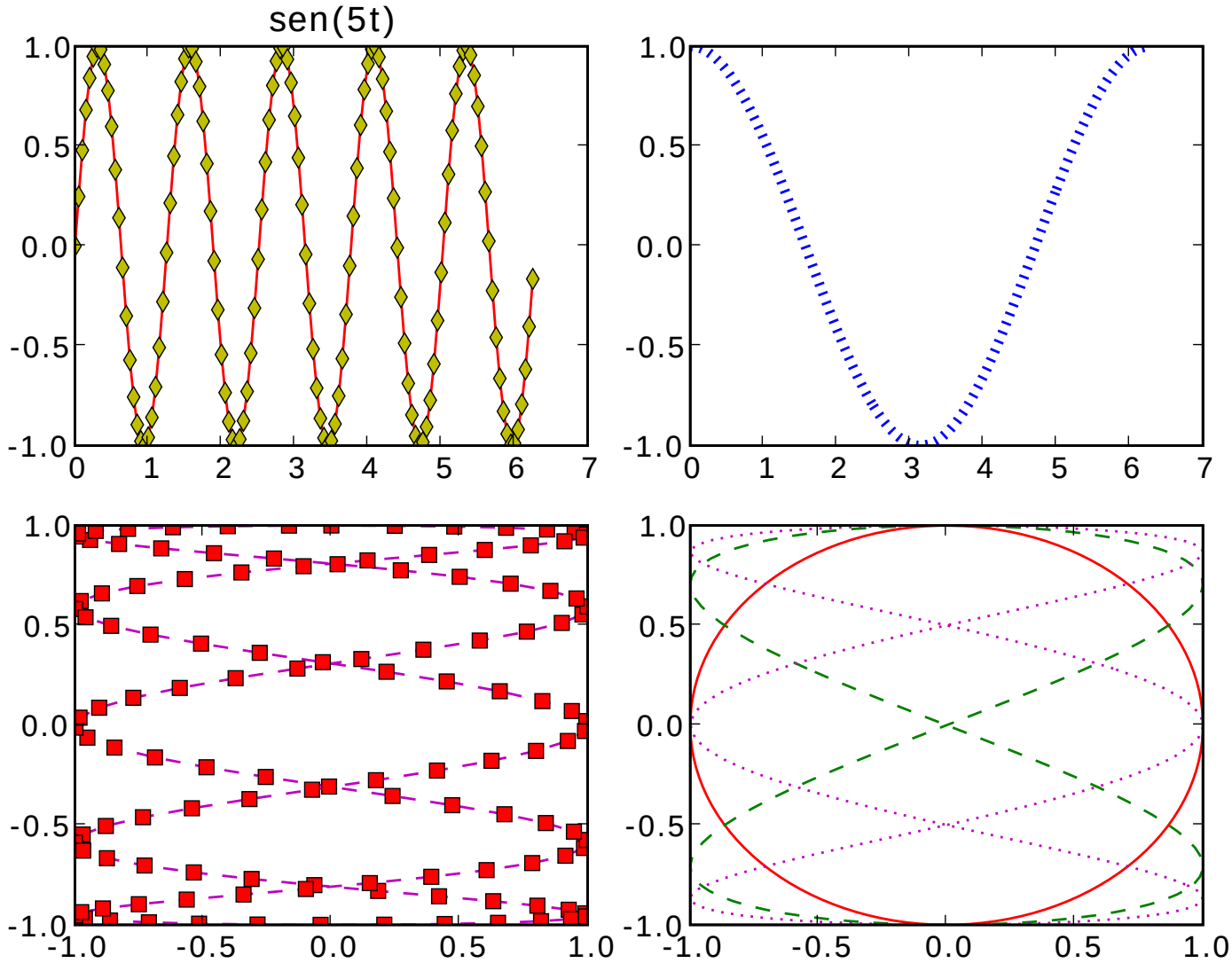
$$x(t) = \sin(i\,t) \qquad\qquad i = 1, 2, \ldots, n$$

Exercise ex2_plot2d.py

```
from pylab import*
import time

t = arange(0,2*pi,0.05)
x = sin (5*t)
y = cos (t)
stx=subplot (2,2,1)
plot (t,x,'r-d',mfc='y')
subplot (2,2,2)
plot (t,y,'b:',lw=4)
sxy=subplot (2,2,3)
plot (x,y,'m--s',mfc='r')

tit=raw_input('Titulo de la
    grafica sin(5*t)=')
stx.set_title(tit)
```

```
n = input ('Dame N (> 1): ')
print type(n)
subplot (2,2,4)
tl=['r-','g--','m:','b-.o']
m=len(tl) #m=4 for this example
for i in range(1,n+1):
    x = sin (i*t)
    resto=i%m
    if (resto==0):
        fl=tl[m-1]
    else:
        fl=tl[resto-1]
    plot (x,y,fl)
    time.sleep(0.3)
show()
```

Ejercicio ej2_plot2d.py



Pandas

- Python tool for data manipulation and analysis (data science).
- It stands for *PANel DAta*.
- Open-source $\rightarrow$ Free/Gratis/Gratis/Frei/Fri.
- It is built on top of *Numpy*.
- Highly optimized: expensive parts in Cython.
- Very well documented.
- Widely used for financial applications.
- Webpage: http://pandas.pydata.org/

Pandas - Some features

- Easy handling of missing data (represented as NaN).
- Size mutability: columns can be inserted and deleted.
- Automatic and explicit data alignment.
- Make it easy to convert Python and NumPy data structures into Pandas objects.
- Intuitive merging and joining data sets.
- Flexible reshaping and pivoting of data sets.
- Hierarchical labeling of axes (possible to have multiple labels per tick).
- IO tools for loading data from flat files (CSV and delimited), Excel files, databases, etc.
- Time series-specific functionality.

Pandas - Resources

- Documentation: http://pandas.pydata.org/pandas-docs/stable/
- Many sources of information:
  - Tutorials.
  - Video tutorials.
  - Online courses.
- Book: Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython by Wes McKinney.

Pandas

- Data structures.
  - *Series*.
  - *DataFrame*.
- Visualization.
- Time series.
- Data reader.

Pandas - Data structures - Series

- One-dimensional indexed (labelled) structure.
- Series(*data*, *index*):
  - *data* can be a list, dictionary, numpy *narray*, etc.
  - *index* is a list of labels (optional, by default $0, 1, \ldots$).
- As *narray*, Series can be viewed, accessed, sliced, compared ($>$, $==$, etc), etc.
- Series handling: *max*, *min*, *sort*, etc.
- Statistics: *mean*, *std*, etc.
- Interoperability with NumPy element-wise math operations.
- Also the dictionary function: *in*, *get*, etc.
- Main difference: Series automatically align the data based on the label.
- Naming the series.
- Example: `/Pandas_examples/1-Series.py`

Pandas - Data structures - DataFrame

- 2-dimensional indexed (labelled) data structure.
- Like a Excel or SQL table.
- DataFrame(*data*, *index*, *columns*):
  - *data* can be a dictionary of lists, dictionaries, narrays or Series.
  - *data* can be a list of dictionaries.
  - *data* can be a 2D narray.
  - *data* can be a Series object.
  - *index* is a list of labels for rows (optional).
  - *columns* is a list of labels for columns (optional).
- The column of a DataFrame object is a Series object.

Pandas - Data structures - DataFrame

- Data alignment is intrinsic: the link between labels and data can not be broken unless done so explicitly.
- Many operations for accessing, addition, selection, alignment, etc.
- Interoperability with NumPy element-wise math operations.
- Operations between *DataFrame* and *Series*: by default, the *Series* index is aligned on the *DataFrame* columns (broadcasting row-wise).
- Higher dimensions: *Panel* (deprecated).
- Example: `/Pandas_examples/2-DataFrame.py`

Pandas - Data structures

- Viewing:
  - *head(n)/tail(n)*: returns the n first/last elements.
  - *index/columns*: returns the index/column of the structure.
  - *describe()*: returns statistical measures (mean, std, quantiles, etc.).
- Getting/Setting:
  - *['C']*: returns the column called *'C'*.
  - *[n:m]*: returns all the columns between n and m (slicing).
  - *loc['r']*: returns the row called *'r'*. Slicing (:) also available.
  - *at['r', 'C']*: returns the value at row *'r'* and *'C'*.
  - *iloc[i]*: same as *loc['r']* but using position i. Slicing (:) also available.
  - *ix['i']*: works with indexes or positions.
- Operations:
  - *mean()*, *std()*, *cumsum()*, T, etc.
  - *apply(f)*: applies f.
- Others:
  - Merging: *concat*, *join*, *append*, etc.
  - Grouping: *groupby*.
  - Reshaping: *stack/unstack*, *pivot_table*

Pandas - Visualization

- Pandas provides advanced visualization tools.
- Based on *Matplotlib*, easier to use.
- Several methods: *line* (line plot), *bar* (bars), *hist* (histograms), *box* (boxplots), *kde* (density), *area* (areas), *scatter* (scatter plots), *hexbin* (hexagonal bins) and *pie* (pie plots).
- The returning value is a *Matplotlib Axes* object.
- Highly customizable (color, legends, size, orientation, scales, etc).
- Other functions for special plots like Andrews curves, scatter matrix, density plot, autocorrelation plot, bootstrap plot, etc.
- *Matplotlib* can be also used (pandas structures act as Numpy arrays).
- Example: `/Pandas_examples/3-Visualization.py`

Pandas - Time series

- Pandas is suitable tool for time series data (financial data).
- Functionalities to:
  - generate sequences of fixed-frequency dates.
  - convert time series to a particular frequency.
  - compute "relative" dates based on various non-standard time increments.
- Based on the *datetime64* type of NumPy.
- Nanosecond precision.
- Main components: *Timestamp* and *Period*.
- List of *Timestamp/Period*: *DatetimeIndex* and *PeriodIndex*.
- Conversion from list-like structures, strings, integers, etc. into *DatetimeIndex*: *to_datetime(list)*.
- Used as indexes in *Series* and *DataFrame* objects.

Pandas - Time series (cont.)

- Generating ranges: *date_range(start, periods, freq)* and *bdate_range(start, periods, freq)*.
- Functionalities:
  - Optimized accessing, slicing, alignment manipulations.
  - Partial indexing: 'year', 'month', etc.
  - Truncation.
- Conversions between Timestamp and Period: *to_period* and *to_timestamp*.
- Many other functionalities: resampling, time zone handling, DateOffsets (implicit and explicit), etc.
- Click to documentation (DateOffsets).
- Example: `/Pandas_examples/4-TimeSeries.py`

Pandas - Data reader

- Functions to extract (financial) data from Internet sources.
- It returns a *DataFrame* object.
- The downloaded data is cached: the subsequent accesses will be faster.
- Currently supported sources (among others): IEX, Quandl, St.Louis FED (FRED), OECD, Eurostat, MOEX, World Bank.
- Useful function: *DataReader(name, source, start, end)*.
- Experimental. Options data from Yahoo! Finance, *Options(name)*.
- Specific requests to avoid the download of all the data: *get_call_data*, *expirity_dates*, etc.
- A lot of information from the World Bank (*wb* package): *search*, *download*, country codes, etc.
- Example: `/Pandas_examples/5-DataReader.py`

Sparse matrices

There are (at least!) two libraries that allow working with sparse matrices:

- `scipy.sparse`
- `pysparse`

SciPy: an example with the sparse library

```
from scipy import sparse
from scipy.sparse.linalg import dsolve as ds
import numpy

n = 5
a = sparse.lil_matrix ((n, n))   # list of lists
for i in range (n-1):
      a[i,i] = 4.0
      a[i,i+1] = -1.0
      a[i+1,i] = -1.0
a[n-1,n-1] = 4.0

x = numpy.array ([1., 2., 3., 4., 5.])
b = a * x
y = ds.spsolve (a,b)   # resolution

aa = a.tocsr ()   # we convert to CSR
bb = aa.matvec (x)
z = ds.spsolve (aa,bb)
```

Examples with pysparse library (I)

```
import numpy
from pysparse import *
n = 5;  a = spmatrix.ll_mat (n,n)
# Symmetric matrix, although we don't profit it
for i in range (n):
    a[i,i] = i+1.0
a[0,0] = 3.0; a[0,3] = -1.0; a[3,0] = -1.0
a[2,4] = 2.0; a[4,2] = 2.0
s = numpy.array ([1.0, 2.0, 3.0, 4.0, 5.0])    # Solution
b = numpy.zeros (n)
a.matvec (s,b)              # b:  second member
x = numpy.zeros (n)

iter = 500;  eps = 0.00001
[par,nit,err] = itsolvers.pcg (a,b,x,eps,iter) # P.C.G.
if (par < 0):
    print ' The CG algorithm has not converged'
else:
    print ' CG converges in %d iter.'  % (nit)
    print ' Achieved relative error:  %12.8e' % (err)
```

Examples with pysparse library (II)

```
n = 5
m = spmatrix.ll_mat (n,n)
for i in range (n):
    m[i,i] = i+1.0
m[0,0] = 3.0; m[0,3] = 1.0; m[3,0] = -1.0
m[2,4] = 2.0; m[4,3] = 2.0     # Non symmetric matrix

c = numpy.zeros (n)
m.matvec (s,c)      # c:  second member
x2 = numpy.zeros (n)
iter = 500
eps = 0.00001
[par,nit,err] = itsolvers.cgs (m,c,x2,eps,iter)
if (par < 0):
    print ' CGS algorithm has not converged'
else:
    print ' CGS converges in %d iterations' % (nit)
    print ' Achieved relative error:  %12.8e' % (err)
```

Examples with pysparse library (III)

```
a2 = a.to_csr ()      # CSR form is required
a2lu = superlu.factorize (a2)      # Factorization

x2 = numpy.zeros (n)
a2.solve (b,x2)
                # Resolution



alu = umfpack.factorize (a)
    # Factorization.  CSR is not required

xlu = numpy.zeros (n)
alu.solve (b,xlu)      # Resolution
```

Examples with pysparse library (IV)

Finite elements assembling:

```
n = 9
a = spmatrix.ll_mat (n,n)
elements = [[3, 0, 4], [1, 4, 0], [4, 1, 5], \
      [2, 5, 1], [6, 3, 7], [4, 7, 3], \
      [7, 4, 8], [5, 8, 4]]
mask = [1, 1, 1]
for ef in elements:
     aelt = numpy.array \
       ([[4., -1., -1.], [-1., 4., -1.], [-1., -1., 4.]])
     a3.update_add_mask (aelt,ef,ef,mask,mask)
```

SciPy `sparse` module

`SciPy` allows seven types of sparse matrices:

1. csc_matrix: "Compressed Sparse Column"
2. csr_matrix: "Compressed Sparse Row"
3. bsr_matrix: block "Sparse Row format"
4. lil_matrix: "LIst of Lists"
5. dok_matrix: ("Dictionnary of Keys")
6. coo_matrix: "COOrdinate" (tuple format)
7. dia_matrix: "DIAgonal"

Sparse matrices building:

- We can efficiently build a sparse matrix in `lil_matrix` format (recommended) or in `dok_matrix` format
- Then, a conversion to CSC or CSR must be performed
  - as `lil_matrix` is row–based, conversion to CSR is more efficient than conversion to CSC
- All conversions from/to CSR, CSC and COO formats are efficient

SciPy `sparse` module

Building a matrix in CSC format:

```
>>> from scipy import sparse
>>> from numpy import array
>>> I = array([0,3,1,0])
>>> J = array([0,3,1,2])
>>> V = array([4,5,7,9])
>>> A = sparse.coo_matrix((V,(I,J)),shape=(4,4))
```

SciPy sparse module

Building a sparse matrix in LIL format and linear system resolution:

```
In [1]: from scipy import sparse, linsolve
In [2]: from numpy import linalg
In [3]: from numpy.random import rand
In [4]: A = sparse.lil_matrix((10, 10))
In [5]: A[0, :5] = rand(10)
In [6]: A[1, 5:10] = A[0, :5]
In [7]: A.setdiag(rand(10))
In [8]: spy(A,marker='.', markersize=8)
In [9]: print A.todense()
```

We convert the matrix to CSR format and solve by the `spsolve` command:

```
In [10]: A = A.tocsr()
In [11]: b = rand(10)
In [12]: x = linsolve.spsolve(A, b)
```

SciPy sparse module

```
# We solve heat equation
# by implicit finite differences
# with a 3x3 mesh

import numpy as np
from pylab import *
import scipy
from scipy import linalg

# number of points in axes
N = 3
D = 4*np.ones (N*N)
T =  -np.ones (N*N)
O =  -np.ones (N*N)
T[N-1::N] = 0.
# Inverse order in NumPy,
TT = T [::-1]

Asp = scipy.sparse.spdiags
([D,O,T,TT,O],[0,-N,-1,1,N],N*N,N*N).tocsr()

# We write the matrix
print Asp.todense()

figure(1)
spy (Asp,marker='.', markersize=8)
show ()

# Right hand side vector
B = np.array([75.,75.,175.,0.,0.,100.,50.,50.,150.])
print B
```

```
# System resolution
C = Asp.tocsc ()
z = scipy.linsolve.spsolve (Asp,B)


z_ = linalg.solve (Asp.todense(),B)

# We check the solution is correct
r = linalg.norm (dot(Asp.todense(),z_)-B)
print r


x = np.linspace (-5,5,N)
y = np.linspace (-5,5,N)
X, Y = np.meshgrid (x, y)
Z = zeros ((N,N))
for i in range (0,N):
    for j in range (0,N):
Z[i,j] = z[i*(N)+j]

# We plot the solution
figure (2)
cset1 = contourf (X, Y, Z, cmap=cm.get_cmap('jet'))
title ('Temperature distribution ')
show ()
```

Excel files read/write

There are different libraries which allow working with Excel files. For example:

`http://www.python-excel.org/`

When we open an Excel file, we are creating an object of the `book` class

- This object has different attributes; for example, its sheets (which are objects of the `sheet` class)
  - Each sheet has, among others, a name (`name`), some rows (`row`), some columns (`col`), numbers of rows (`nrows`) and columns (`ncols`), ...

Excel files read/write

```
1  from mmap import mmap
2  from xlrd import open_workbook
3
4  wb = open_workbook ('test.xls')          # We open the book
5
6  data = []                  # We create a list to store the data
7  s = wb.sheets()[0]     # We select the first sheet
8
9  print
10 print ' Sheet: ', s.name
11 for row in range (s.nrows):      # We go through the rows
12     print ' Row: ', row
13     values = []
14     for col in range (s.ncols): # We go through the columns
15         print '    Column: ', col
16         values.append (s.cell(row,col).value)
17                 # We read the value in the cell and store it
18     data.append (values)
```

Excel files read/write

```
1  from tempfile import TemporaryFile
2  from xlwt import Workbook, Formula
3
4  book = Workbook ()
5  sheet1 = book.add_sheet ('Sheet1')
6  book.add_sheet ('Hoja2')
7
8  sheet1.write (0,0,'A1')
9  sheet1.write (0,1,'B1')
10 row1 = sheet1.row (1)
11 row1.write (0,'A2')
12 row1.write (1,'B2')
13 sheet1.col (0).width = 10000
14 a = 3.14
15 b = 2
16 sheet1.write (3, 0, a)
17 sheet1.write (3, 1, b)
18 sheet1.write (3, 2, Formula ("$A$4*$B$4"))
```

Excel files read/write

```
1  sheet2 = book.get_sheet (1)
2  sheet2.row (0).write (0, 'Sheet 2 A1')
3  sheet2.row (0).write (1, 'Sheet 2 B1')
4  sheet2.flush_row_data ()
5  sheet2.write (1, 0, 'Sheet 2 A3')
6  sheet2.col (0).width = 5000
7  sheet2.col (0).hidden = True
8
9  book.save ('simple.xls')
10 book.save (TemporaryFile ())
```

Application: e-mail sending

If we want to send an e-mail, we must:

- connect with the mail server
  - providing a *login* and a *password*
- define the recipient
- define the body and the subject of the message
- eventually, associate an attached file
- send the message (and the attached file)

Python provides some libraries that facilitate these steps

Application: e-mail sending

```
1  import getpass
2
3  print
4  my_password = getpass.getpass ('  Introduce the password: ')
5  print
6
7  # Connection with the UDC mail server.
8
9  import smtplib
10
11 mailServer = smtplib.SMTP ('smtp.udc.es', 25)
12
13 mailServer.ehlo ()
14 mailServer.starttls ()
15 mailServer.ehlo ()
16
17 # Address from we send the message(s).
18
19 mailServer.login ("my_address@udc.es", my_password)
```

Application: e-mail sending

```
# Body of the message.

from email.MIMEMultipart import MIMEMultipart
from email.MIMEText import MIMEText

message = u"""Here, we write the body of the message.
We can include line breaks, and non standard characters (
    spanish 'tildes', e\~nes, ...)"""

# We declare a MIME multipart message.

mensaje = MIMEMultipart ()
mensaje['From'] = "my_address@udc.es"
mensaje['To'] = recipients[i].encode ('latin1')
mensaje['Subject'] = u"Subject".encode ('latin1')

texto = mensaje
coding = 'latin-1'
mensaje.attach (MIMEText(texto.encode('latin-1'), _charset='
    latin-1'))

# Sending.
mailServer.sendmail ("my_address@udc.es", recipients[i],
    mensaje.as_string())
mailServer.close ()
```

Application: e-mail sending

Exercise:

- read a text file that contains (at least) the name, sex, identity number and e-mail addresses of a group of persons:

Name0	M	12345670	address0@udc.es
Name1	F	12345671	address1@udc.es
Name2	F	12345672	address2@udc.es
Name3	M	12345673	address3@udc.es

- send a customized e-mail to each of the previous persons