INSTITUT
de MATHEMATIQUES
de TOULOUSE

Python pour les mathématiques numériques et l'analyse de données

Laurent Risser
Ingénieur de Recherche à l'Institut de Mathématiques de Toulouse

lrisser@math.univ-toulouse.fr

En bref :

- Libre
- Langage très compact
- Nombreuses extensions et bibliothèques disponibles
- Efficace en mathématiques numériques grâce à la bibliothèque NumPy
- Orienté objet (mais on peut en faire abstraction)
- Grande communauté très active

Installation :

- Voir www.python.org/getit/ , www.scipy.org/install.html , scikit-learn.org/stable/install.org
- Distributions « tout intégré » comme enthought canopy ou anaconda faciles à installer

Deux manières principales d'utiliser Python :

- Interprétation de commandes, *e.g.* IDLE (environnement de développement intégré) ou iPython
- En tant que script

Remarque :

- Plusieurs *releases* de Python sont couramment utilisées : les 2.6, 2.7 et 3.3
- Pour les mathématiques numériques, la 2.7 est la plus courante (utilisée dans ce tutorial)

Un 1er exemple :

1) Sous IDLE ou ipython

```
blaye:tmp risser$ python
Python 2.7.2 (default, Oct 11 2012, 20:14:37)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Ap
Type "help", "copyright", "credits" or "licens
>>> 1+1
2
>>>
```

```
blaye:tmp risser$ ipython
Python 2.7.2 (default, Oct 11 2012, 20:14:3
Type "copyright", "credits" or "license" fo

IPython 1.1.0 -- An enhanced Interactive Py
?         -> Introduction and overview of I
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'o

In [1]: 1+1
Out[1]: 2

In [2]:
```

2) En tant que script

```
blaye:tmp risser$ ls
test.py
blaye:tmp risser$ more test.py
a=1+1

print a

blaye:tmp risser$ python test.py
2
```

3) En tant que fichier compilé

→ Simplification de la distribution des éxécutables

→ Avec PyInstaller, cx_Freeze (Unix, Windows, Mac), Py2app (Mac), Py2exe (Windows)

Plan

1) Présentation du langage Python (1h10)

- Types de données
- Structures de contrôle
- Fonctions
- Classes
- Modules

2) Utilisation des bibliothèques Numpy/Matplotlib/Scipy (1h10)

- Numpy et les arrays
- Matplotlib
- Scipy

3) Utilisation de la bibliothèque scikit-learn (10 min)

Basic data types : integer, float, boolean, string

Pas de déclaration de variables :
→ Taper `a=3` défini « a » comme un entier (integer) et lui donne la valeur 3
→ Taper `b=1.` défini « b » comme un flottant (float) et lui donne la valeur 1

```
a+1        → 4
a-1        → 2
a*2        → 6
a**2       → 9
pow(a,2)   → 9
```

Langage fortement typé :

```
a/2        → 1
float(a)/2 → 1.5
a/2.       → 1.5
```

Remarque :
- Exemple lancé sous Python 2.7
- Dans Python 3.3, le résultat est un float

Comparaison faite par `==` , `>` , `<` , `!=` (ou `<>`)
→ retourne un bouléen True (1) ou False (0)

Informations sur une variable :
→ Valeur : `print a` → 3
→ Type : `type(a)` → int

Basic data types : integer, float, boolean, string

Chaines de caractères (strings) :
→ utiliser les guillemets simples ou doubles pour les strings

```
a='hello'
b=" world"
a+b
  → 'hello world'
print a+b
  → hello world
```

Basic data structures : List, Tuple, Dictionary

liste :

- combinaison de *basic data types*
- entouré de crochets [ ]

```
liste_A = [0,3,2,'hi']
liste_B = [0,3,2,4,5,6,1]
liste_C = [0,3,2,'hi',[1,2,3]]
```

Pour accéder à un élément de la liste utiliser des crochets (Le 1er indice est 0 !)

```
Liste_A[1] → 3
```

Pour acceder au dernier élément de la liste : taper l'index -1

```
liste_C[-1]     →  [1, 2, 3]
liste_C[-1][0]  →  1
liste_C[-2]     →  'hi'
```

Extraire une sous-liste :

```
liste_B[0:2]    →  [0, 3]
liste_B[0:5:2]  →  [0, 2, 5]
liste_B[::-1]   →  [1, 6, 5, 4, 2, 3, 0]
```

start:stop:step !!!

Basic data structures : List, Tuple, Dictionary

Remarque importante : pour une liste, la commande = marche par référence

```
L1=[1,2,3]
L2=L1
L3=L1[:]
L1[0]=10

L1   →   [10, 2, 3]
L2   →   [10, 2, 3]
L3   →   [1, 2, 3]
```

Pour copier une liste qui contient des listes, utiliser un 'deep copy'
→ `import copy` puis `newList=copy.deepcopy(mylist)`

Quelques fonctions intéressantes pour les listes :

```
List=[3,2,4,1]
List.sort()  →    [1, 2, 3, 4]
List.append('hi') →    [1, 2, 3, 4,'hi']
List.count(3)    →    1
List.extend([7,8,9])    →          [1, 2, 3, 4, 'hi', 7, 8, 9]
List.append([10,11,12])    →    [1, 2, 3, 4,'hi', 7, 8, 9, [10, 11, 12]]
```

Basic data structures : List, Tuple, Dictionary

Tuples :

- Même chose qu'une liste mais ne peux pas être modifié
- Défini avec des parenthèses

```
MyTuple=(0,3,2,'h')
MyTuple[1]      →   3
MyTuple[1]=10   →   TypeError: 'tuple' object does not support item assignment
```

Basic data structures : List, Tuple, Dictionary

Dictionaires :

- Similaire à une liste mais chaque entrée est assignée par une clé/un nom
- Défini avec des {}

```
months = {'Jan':31 , 'Fev': 28, 'Mar':31}
    months['Jan']       →    31
    months.keys()       →    ['Jan', 'Fev', 'Mar']
    months.values()     →    [31, 28, 31]
    months.items()      →    [('Jan', 31), ('Fev', 28), ('Mar', 31)]
```

Les blocs de codes sont définis par deux points suivi d'une indentation fixe.
→ intérêt : forcer à écrire du code facile à lire

Structures de contrôle :
- if: elif: else:
- for *var* in *set*:
- while *condition*:

```
a=2
if a>0:
        b=0
        for i in range(4):
                b=b+i
else:
        b=-1

print b
        → -1
```

```
for i in range(4):
                print i
→ 0
→ 1
→ 2
→ 3
```

```
for i in range(1,8,2):
                print i
→ 1
→ 3
→ 5
→ 7
```

Fonctions sont définies par:

```
def FunctionName(args):
        commands
        return value
```

→ *return* est optionnel (None retourné si rien).
→ On peut retourner plusieurs valeurs en utilisant des virgules (un tuple est retourné)

```
def pythagorus(x,y):
      """ Computes the hypotenuse of two arguments """
      r = pow(x**2+y**2,0.5)
      return x,y,r

pythagorus(3,4)          →    (3, 4, 5.0)
pythagorus(x=3,y=4)      →    (3, 4, 5.0)
pythagorus(y=4,x=3)      →    (3, 4, 5.0)
help(pythagorus)         →    Computes the hypotenuse of two arguments
pythagorus.__doc__       →    'Computes the hypotenuse of two arguments'
```

Valeurs par défaut, ici avec : `def pythagorus(x=1,y=1):`

```
pythagorus()             →    (1, 1, 1.4142135623730951)
```

Python est un langage orienté objet. Il est alors possible d'y définir des classes :

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

        def functionname(self,args):
                ...
```

- La classe *ClassName* hérite de la classe *superclass* (optionnellement).
- *__init__* est le constructeur de la classe
- *functionname* est une fonction de la classe

A propos des *self :*
- Peuvent être ignorées dans les fonctions
- Doivent être le 1er argument de chaque fonction

Pour utiliser une fonction de la classe :

```
var = ClassName ()
var.functionname()
```

Remarque : Toutes les fonctions d'une classe Python sont publiques et aucune n'est privée.

→ Un module contient plusieurs fonctions et commandes
→ Fonctions et commandes regroupées dans un fichier *.py*
→ Module appelé en utilisant `import`

toto.py :

```
def SayHi():
    print 'Hi'

def DivBy2(x):
    return x/2.
```

```
import toto

toto.SayHi()
→ Hi

print toto.DivBy2(10)
→ 5.0
```

```
import toto as tt

tt.SayHi()
→ Hi

print tt.DivBy2(10)
→ 5.0
```

```
from toto import SayHi

SayHi()
→ Hi

print DivBy2(10)
→ error
```

```
from toto import *

SayHi()
→ Hi

print DivBy2(10)
→ 5.0
```

Module considéré comme un script lorsqu'il contient des commandes :
- Lors de l'import d'un script, les commandes vont être exécutées
- Lors de l'import de fonctions, elles sont juste chargées en mémoire

Compilation du module :
- Lors de son 1er appel, un module python est compilé dans un fichier *.pyc*
- Le fichier compilé est utilisé lors des appels suivants
- Pour recharger (et recompiler) un module il faut taper la commande `reload(name)`

Répertoires des modules :
- Listés dans `sys.path` (après un `import sys`)
- Ajout avec `sys.path.append('mypath')`

→ Les packages regroupent plusieurs modules dans un (des) répertoire(s) donné(s)

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
            __init__.py
            wavread.py
            wavwrite.py
            aiffread.py
            aiffwrite.py
            ...
      effects/                  Subpackage for sound effects
            __init__.py
            echo.py
            surround.py
            ...
      ...
```

http://docs.python.org/2/tutorial/modules.html

Pour charger un sous-module : `import sound.effects.echo` ou `from sound.effects import echo`

Fichiers *__init__.py* :

- Pour que Python traite le répertoire comme un package
- Peuvent être vide, contenir des commandes à exécuter, ou définir des variables

→ Gestion des arguments si le .py est un script avec `argv`

MyScript.py :

```
import sys
print "Name of the script:" , sys.argv[0]
argc=len(sys.argv)
print "Number of arguments:" , argc
print "Arguments are:" , str(sys.argv)
```

python MyScript.py (dans le shell)

→ Name of the script: MyScript.py
→ Number of arguments: 1
→ Arguments are: ['MyScript.py']

python MyScript.py toto 1 2.3 (dans le shell)

→ Name of the script: MyScript.py
→ Number of arguments: 4
→ Arguments are: ['MyScript.py' , 'toto' , '1' , '2.3']

Remarque : Les arguments ne sont pas acceptés si le .py est chargé avec import (dans un autre script ou sous IDLE/ipython)

Quelques notes pour finir cette partie :

Note 1 : Python est *case sensitive*, c'est a dire qu'une variable *a* est différente de la variable *A*

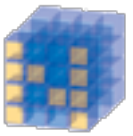
Note 2 : commentaires avec `#`

Note 3 : appel système avec `import os` puis `os.system('...')`

Note 4 : Quitter Python avec *ctrl+d* ou bien `quit()`

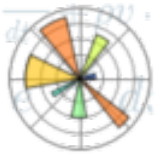
Intérêt de NumPy, Matplotlib et SciPy :
→ Faire du calcul scientifique et de l'analyse de données sous Python

NumPy :



- Contient le *data type* array et des fonctions pour le manipuler
- Contient aussi quelques fonctions d'algèbre linéaire et statistiques
- Supporté par Python 2.6 et 2.7, ainsi que 3.2 et plus récent

Matplotlib :



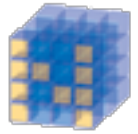
- Fonctions de visualisation / graphs
- Commandes proches de celles sous matlab (mais un résultat plus sympa)
- Aussi connu sous le nom de pylab

SciPy :

- Modules d'algèbre linéaire, statistique et autre algorithmes numériques.
- Quelques fonctions redondantes avec celles de NumPy.

- Fonctions de SciPy plus évolués que celles de NumPy.
- Celles de NumPy sont là pour assurer la compatibilité avec du vieux code.
- NumPy ne devrait être utilisé que pour ses arrays

2.1.1) Structure Array

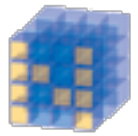


NumPy

- De loin, la structure de donnée la plus utilisée pour les maths numériques sous Python
- Tableau en dimension *n = 1, 2, 3, … , 40*
- Tous les éléments on le même type (booléen, entier, réel, complexe)

```
import numpy as np

my_1D_array = np.array([4,3,2])
print my_1D_array

→    [4 3 2]

my_2D_array = np.array([[1,0,0],[0,2,0],[0,0,3]])
print my_2D_array

→    [[1 0 0]
      [0 2 0]
      [0 0 3]]
```

```
myList=[1,2,3]
my_array = np.array(myList)
print my_array
→    [1 2 3]


a=np.array([[0,1],[2,3],[4,5]])

a[2,1]
→    5

a[:,1]
→    array([1, 3, 5])
```



NumPy

2.1.2) Fonctions de construction d'Arrays

- arange() : génère un array de la même manière que range pour une liste

```
np.arange(5)        ↔   np.array([0, 1, 2, 3, 4])
np.arange(3,7,2)    ↔   np.array([3, 5])
```

- ones(), zeros() : génère un array ne contenant que des 1 ou des zéros

```
np.ones(3)          ↔   np.array([1., 1., 1.])
np.ones((3,4))      ↔   np.array([[1., 1., 1., 1.],[1., 1., 1., 1.],[1., 1., 1., 1.]])
```

- eye() : génère une matrice identité

```
np.eye(3)   ↔   np.array([[1.,0., 0.],[0., 1., 0.],[0., 0., 1.]])
```

- linespace(start, stop, spacing) : produit un vecteur avec des éléments espacés linéairement

```
np.linspace(3, 7, 3)    ↔   np.array([3., 5., 7.])
```

- mgrid() et ogrid() : similaire à meshgrid dans matlab

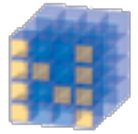
```
m=np.mgrid[0:3,0:2]
m → array([ [[0, 0], [1, 1], [2, 2]] ,
            [[0, 1], [0, 1], [0, 1]] ])
```

```
o=np.ogrid[0:3,0:2]
o → [  array([[0],[1], [2]]) ,  array([[0, 1]])  ]
```

- Matrices aléatoires avec `from numpy.random import *` :

`rand(size)`, `randn(size)`, `normal(mean,stdev,size)`, `uniform(low,high,size)`, `randint(low,high,size)`

2.1.3) Autres fonctions utiles pour les Arrays



NumPy

```
a=np.array([[0,1],[2,3],[4,5]])
np.ndim(a)            →   2
np.size(a)            →   6
np.shape(a)           →   (3, 2)
np.transpose(a)       →   array([[0, 2, 4],[1, 3, 5]])
a.min(), np.min(a)    →   0
a.sum(), np.sum(a)    →   15
a.sum(axis=0)         →   array([6, 9])
a.sum(axis=1)         →   array([1, 5, 9])
```

(Nombre de dimensions)
(Nombre d'éléments)
(Tuple contenant la dimension de *a*)
(Transposé)
(Valeur min)
(Somme des valeurs)
(Somme sur les colonnes)
(Somme sur les lignes)

mais aussi max, mean, std, …, et encore :

```
np.r_[1:4,10,11]          →   array([1, 2, 3, 10, 11])
np.c_[1:4,11:14]          →   array([1, 2, 3, 0, 4])
np.c_[1:4,11:15]          →   array dimensions must agree
np.arange(6).reshape(3,2) →   array([[0,1],[2,3],[4,5]])
```

(Concaténation en ligne)
(Concaténation en colonne)

(modification de la forme)

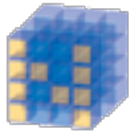
Remarque importante : = fonctionne par référence. La copie d'un array est alors c=a.copy()

```
A1=np.array([1,2,3])
A2=A1
A3=A1.copy()
A1[0]=10
```

→

```
A1   →   array( [10, 2, 3] )
A2   →   array( [10, 2, 3] )
A3   →   array( [1, 2, 3] )
```

2.1.4) Contrôler le type des Arrays



NumPy

→ On peut spécifier le type des valeurs d'un array
→ Il en existe bien plus que dans Python standard, *e.g.* :

```
Bool_, int8, int16, int32, int64, uint8, ..., uint64, float16, ..., float64, complex64
```

Exemples de déclaration :

```
a=np.ones((2,4),dtype=float)

→    array([[ 1.,  1.,  1.,  1.],
            [ 1.,  1.,  1.,  1.]])
```

```
b=np.float32(1.0)

→    1.0
```

```
c=np.int_([1,2,4])

→    array([1, 2, 4])
```

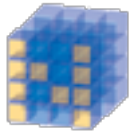
Convertir le type d'un array :

```
z = np.arange(3, dtype=np.uint8)
z.astype(float)
np.int8(z)
```

Connaitre le type d'un array :

```
z.dtype                    →    dtype('uint8')
```

2.1.5) Opérations de base sur les Arrays



NumPy

On considère :

```
a=np.arange(6).reshape(3,2)
→    array([[0, 1],
            [2, 3],
            [4, 5]])
```

```
b=np.arange(3,9).reshape(3,2)
→    array([[3, 4],
            [5, 6],
            [7, 8]])
```

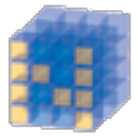
```
c=np.transpose(b)
→    array([[3, 5, 7],
            [4, 6, 8]])
```

Alors :

```
a+b
→    array([[3, 5],
            [7, 9],
            [11, 13]])
a*b
→    array([[0, 4],
            [10, 18],
            [28, 40]])
np.dot(a,c)
→    array([[4, 6, 8],
            [18, 28, 38],
            [32, 50, 68]])
```

```
np.power(a,2)
→    array([[0, 1],
            [4, 9],
            [16, 25]])
np.power(2,a)
→    array([[1, 2],
            [4, 8],
            [16, 32]])
a/3
→    array([[0, 0],
            [0, 1],
            [1, 1]])
```

2.1.6) Lecture/écriture de fichier csv contenant un array



NumPy

Il existe une *basic data structure* **File** :

`input = open('filename')` , `input.close()` ,`readlines()` , `writelines()` , `readline()` , `writeline()` , `read()` , `write()`

… mais le plus simple est d'utiliser les fonctions numpy adaptées :

```
from numpy import genfromtxt
from numpy import savetxt

data = genfromtxt('testFile.csv', delimiter=',')
data
→ array([[ 1.,  2.,  3.],
         [ 4.,  5.,  6.],
         [ 7.,  8.,  9.]])

data2=data*2

savetxt('OutputFile.csv',data2,delimiter=',')
```

Fichier testFile.csv :

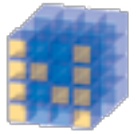
1 , 2 , 3
4 , 5 , 6
7 , 8 , 9

Fichier OutputFile.csv :

2.0e+00 , 4.0e+00 , 6.0e+00
8.0e+00 , 1.0e+01 , 1.2e+01
1.4e+01 , 1.6e+01 , 1.8e+01

Note : Il est aussi intéressant de jeter un oeuil au module csv et au module pickle

2.1.7) Fonction lambda sur un array



NumPy

→ Une fonction lambda peut être pratique pour définir les valeurs d'un array, *eg* :

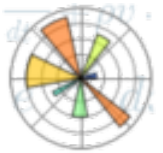
```
import numpy as np

gaussian = lambda x: np.exp(-x**2/1.5)
x=np.arange(-2,1.5,0.5)
y=gaussian(x)
```

```
x    →    array([-2.    , -1.5   , -1.    , -0.5    ,  0. ,  0.5    , 1.    ])
y    →    array([0.0694,  0.2231,  0.5134 ,  0.8464,  1.  ,  0.8464,  0.5134])
```

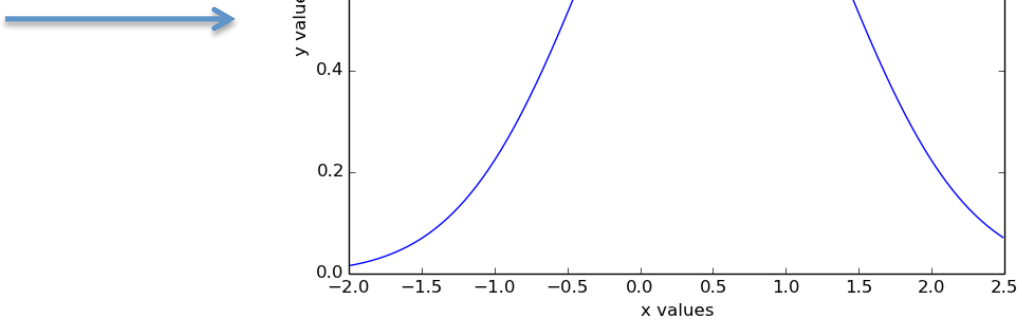
```
sumTimes2= lambda x,y: (x+y)*2
a=np.arange(0,2,0.5)
b=np.arange(10,12,0.5)
c=sumTimes2(a,b)
```

```
a    →    array([ 0. ,  0.5,  1. ,  1.5])
b    →    array([ 10. ,  10.5,  11. ,  11.5])
c    →    array([ 20.,  22.,  24.,  26.])
```

→ Ecrit pour ressembler aux fonctions de graph 2D de Matlab
→ On appel la Matplotlib avec : `import pylab`

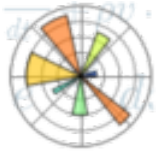


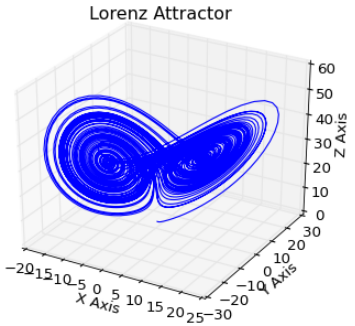

```
import numpy as np
from pylab import *

gaussian = lambda x: np.exp(-(0.5-x)**2/1.5)
x=np.arange(-2,2.5,0.01)
y=gaussian(x)

plot(x,y)
xlabel('x values')
ylabel('y values')
title('Gaussian function')
show()
```

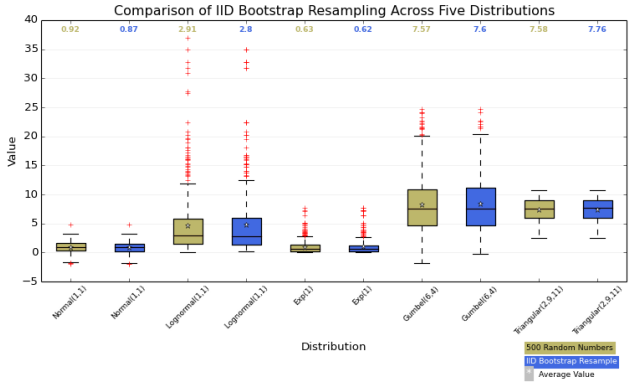


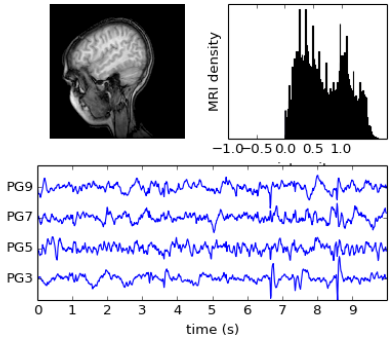
Remarque :
→ Pour s'assurer que le plot ne se ferme pas à la fin d'une fonction : utiliser `show()`

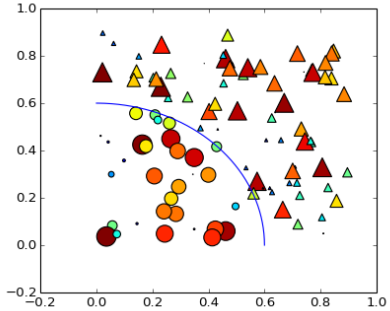
Pour se faire une idée de ce que l'on peut faire avec MatplotLib:
http://matplotlib.org/1.3.1/gallery.html

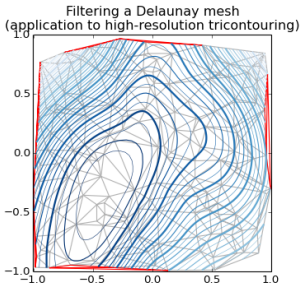


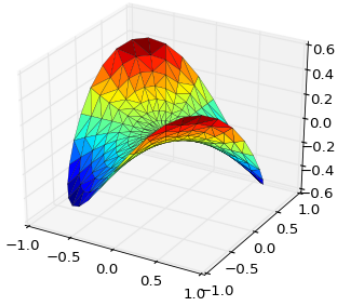


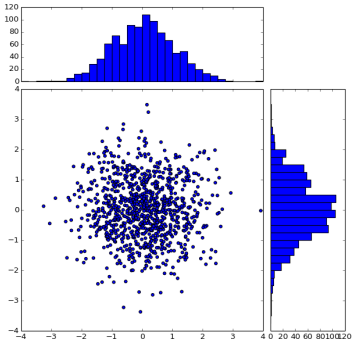


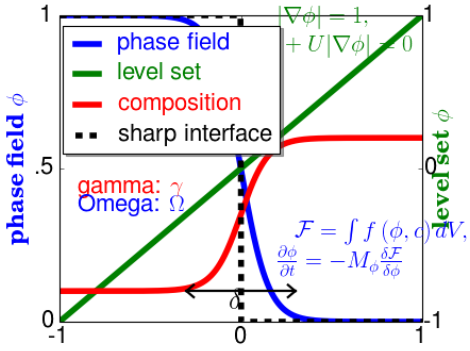


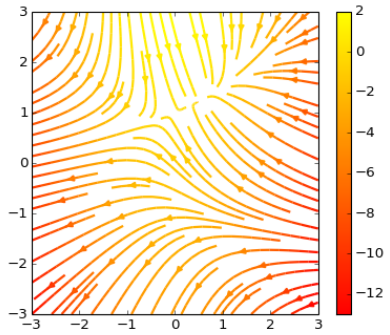










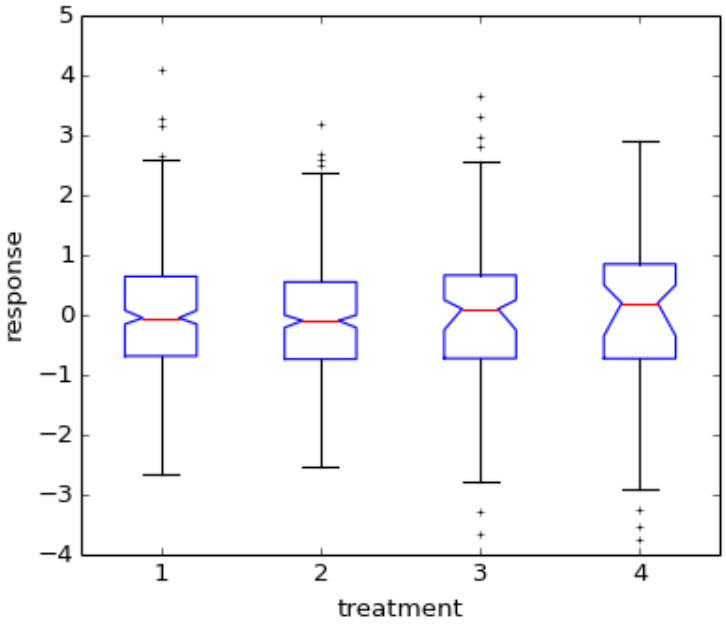


…

→ On trouve le code python corresondant à chaque graph sur la page

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

def fakeBootStrapper(n):
        if n == 1:
                med = 0.1
                CI = (-0.25, 0.25)
        else:
                med = 0.2
                CI = (-0.35, 0.50)
        return med, CI

np.random.seed(2)
inc = 0.1
e1 = np.random.normal(0, 1, size=(500,))
e2 = np.random.normal(0, 1, size=(500,))
e3 = np.random.normal(0, 1 + inc, size=(500,))
e4 = np.random.normal(0, 1 + 2*inc, size=(500,))
treatments = [e1,e2,e3,e4]
med1, CI1 = fakeBootStrapper(1)
med2, CI2 = fakeBootStrapper(2)
medians = [None, None, med1, med2]
conf_intervals = [None, None, CI1, CI2]
```

```
...
fig, ax = plt.subplots()
pos = np.array(range(len(treatments)))+1
bp=ax.boxplot(treatments, sym='k+', positions=pos,
        notch=1,bootstrap=5000, usermedians=medians,
        conf_intervals=conf_intervals)

ax.set_xlabel('treatment')
ax.set_ylabel('response')
plt.setp(bp['whiskers'], color='k', linestyle='-' )
plt.setp(bp['fliers'], markersize=3.0)
plt.show()
```



SciPy est une bibliothèque pour les mathématiques, la science, et l'ingénierie
→ Utilise massivement la structure array de NumPy.

On peut voir ce que Scipy peut faire à la page *http://docs.scipy.org/doc/scipy/reference/* :

- Integration (scipy.integrate)
- Optimization and root finding (scipy.optimize)
- Interpolation (scipy.interpolate)
- Fourier Transforms (scipy.fftpack)
- Signal Processing (scipy.signal)
- Linear Algebra (scipy.linalg)
- Sparse Eigenvalue Problems with ARPACK
- Compressed Sparse Graph Routines scipy.sparse.csgraph
- Spatial data structures and algorithms (scipy.spatial)
- Statistics (scipy.stats)
- Multi-dimensional image processing (scipy.ndimage)
- Clustering package (scipy.cluster)
- Orthogonal distance regression (scipy.odr)
- Sparse matrices (scipy.sparse)
- Sparse linear algebra (scipy.sparse.linalg)
- Compressed Sparse Graph Routines (scipy.sparse.csgraph)
- File inputs/outputs (scipy.io)
- Weave – C/C++ integration (scipy.weave)

Exemple de fonctions d'algèbre linéaire :


```
import numpy as np
from scipy import linalg

A = np.array([[1,2],[3,4]])

linalg.inv(A)
→     array([[-2. , 1. ], [ 1.5, -0.5]])

linalg.pinv(A) #generalized inverse
→     array([[-2. ,  1. ],[ 1.5, -0.5]])

A.dot(linalg.inv(A)) #double check
→     array([[1.0e+00, 0.0e+00],[4.4e-16, 1.0e+00]])

linalg.det(A)
→     -2.0
```

```
la,v = linalg.eig(A)
l1,l2 = la

→     print l1, l2 #eigenvalues
(-0.372281323269+0j) (5.37228132327+0j)

print v[:,0] #first eigenvector
→     [-0.82456484 0.56576746]

print v[:,1] #second eigenvector
→     [-0.41597356 -0.90937671]
```

```
…
U,s,Vh = linalg.svd(A) #singular value decomp.
…
linalg.expm2(A)         #matrix exponential
…
```

Remarque : certaines fonctions sont aussi disponibles dans np.linalg mais sont plus simples

Exemple de fonction d'optimisation :


```
import numpy as np
from scipy.optimize import minimize

def rosen(x):
    """The Rosenbrock function"""
    return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)

x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
rosen(x0)
→   848.2200

res = minimize(rosen, x0, method='nelder-mead',options={'xtol': 1e-8, 'disp': True})

→   Optimization terminated successfully. Current function value: 0.000000
→   Iterations: 339 Function evaluations: 571

print(res.x)
→   [ 1. 1. 1. 1. 1.]

rosen(res.x)
→   4.861e-17
```

Exemple de fonction en statistique :

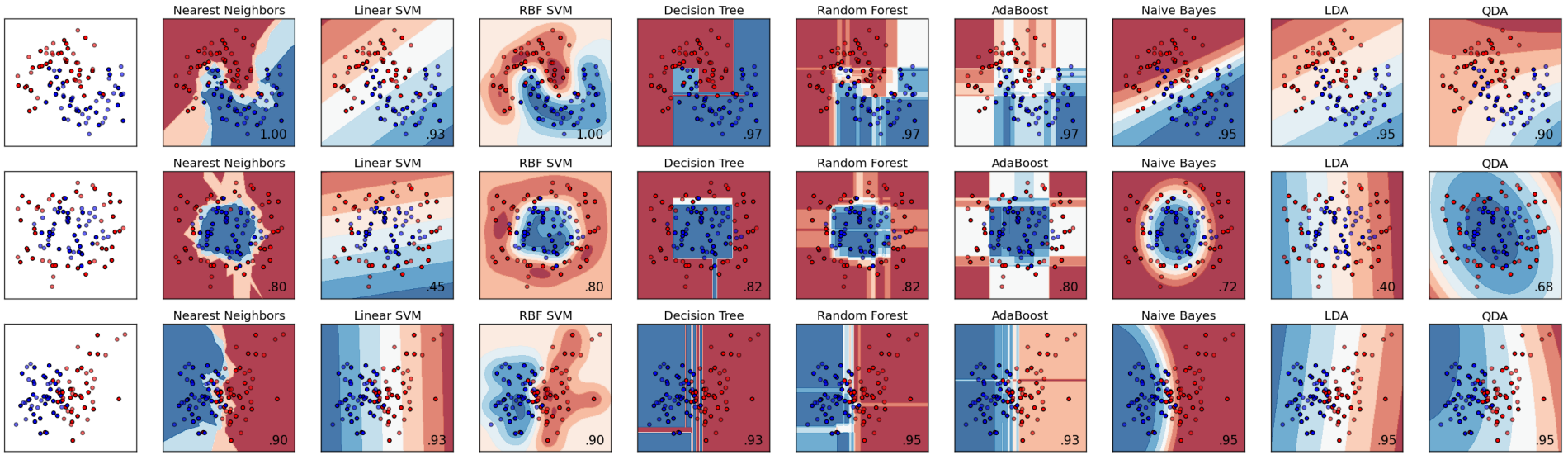

```
import scipy.stats

rvs1 = scipy.stats.norm.rvs(loc=5, scale=10, size=500)
rvs2 = scipy.stats.norm.rvs(loc=5, scale=10, size=500)
rvs3 = scipy.stats.norm.rvs(loc=8, scale=10, size=500)

scipy.stats.ttest_ind(rvs1, rvs2) #t-test (returns: calculated t-statistic / two-tailed p-value)
→    (-0.5489, 0.5831)

scipy.stats.ttest_ind(rvs1, rvs3)
→    (-4.533, 6.507e-6)

scipy.stats.ks_2samp(rvs1, rvs2) #Kolmogorov-Smirnov test (returns: KS statistic / two-tailed p-value)
→    (0.0259, 0.9954)

scipy.stats.ks_2samp(rvs1, rvs3)
→    (0.1139, 0.0027)
```

3) Scikit-learn

Scikit-learn est une bibliothèque pour le data mining et l'analyse de données sous Python.
→ Utilise massivement NumPy, Matplotlib et SciPy.

On peut se faire une idée de ce que Scikit-learn peut faire à *http://scikit-learn.org/stable/* :

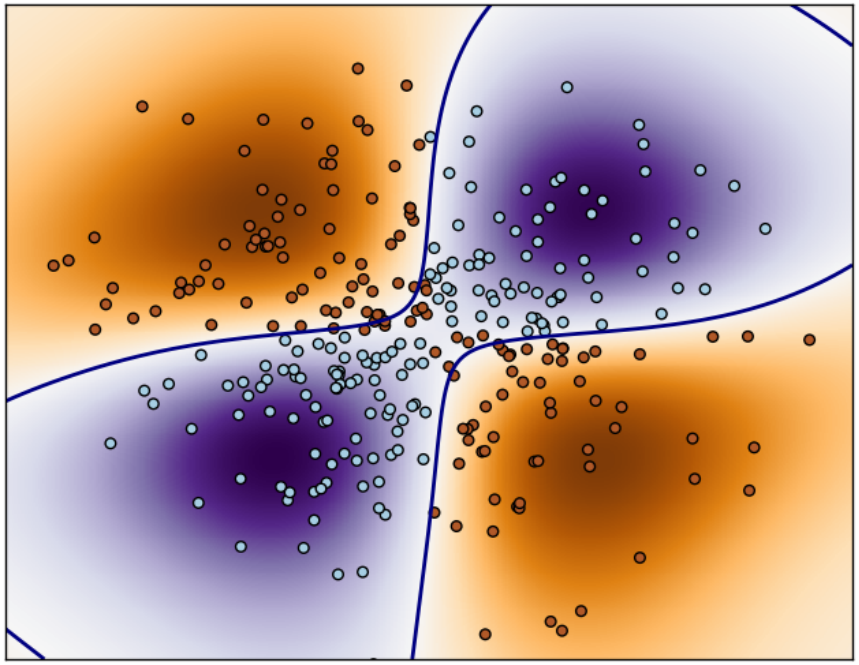
- Classification → SVM, nearest neighbors, random forest, ...
- Regression → SVR, ridge regression, Lasso, ...
- Clustering → k-Means, spectral clustering, mean-shift, ...
- Dimensonality reduction → PCA, Isomap, non-negative matrix factorization.
- Clustering → grid search, cross validation, metrics.



Exemple 1 : Non-linear SVM (http://scikit-learn.org/stable/auto_examples/svm/plot_svm_nonlinear.html)

```
import numpy as np
import pylab as pl
from sklearn import svm

xx, yy = np.meshgrid(np.linspace(-3, 3, 500),
                     np.linspace(-3, 3, 500))
np.random.seed(0)
X = np.random.randn(300, 2)
Y = np.logical_xor(X[:, 0] > 0, X[:, 1] > 0)

# fit the model
clf = svm.NuSVC()
clf.fit(X, Y)
# plot the decision function for each datapoint on the grid
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

...
```

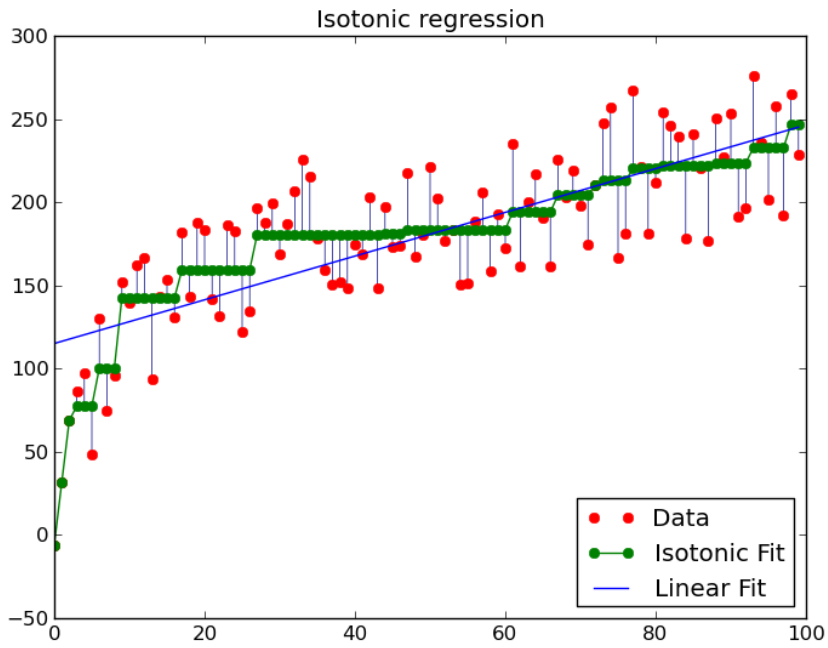
```
...

pl.imshow(Z, interpolation='nearest', extent=(xx.min(), xx.max(), yy.min(),
                yy.max()), aspect='auto', origin='lower', cmap=pl.cm.PuOr_r)
contours = pl.contour(xx, yy, Z, levels=[0], linewidths=2, linetypes='--')
pl.scatter(X[:, 0], X[:, 1], s=30, c=Y, cmap=pl.cm.Paired)
pl.xticks(())
pl.yticks(())
pl.axis([-3, 3, -3, 3])
pl.show()
```



Exemple 2 : isotonic regression (http://scikit-learn.org/stable/_downloads/plot_isotonic_regression.py)

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
from sklearn.linear_model import LinearRegression
from sklearn.isotonic import IsotonicRegression
from sklearn.utils import check_random_state

n = 100
x = np.arange(n)
rs = check_random_state(0)
y = rs.randint(-50, 50, size=(n,)) + 50. * np.log(1 + np.arange(n))

# Fit IsotonicRegression and LinearRegression models
ir = IsotonicRegression()
y_ = ir.fit_transform(x, y)
lr = LinearRegression()
lr.fit(x[:, np.newaxis], y)    # x needs to be 2d for LinearRegression

# plot result
segments = [[[i, y[i]], [i, y_[i]]] for i in range(n)]
lc = LineCollection(segments, zorder=0)
lc.set_array(np.ones(len(y)))
lc.set_linewidths(0.5 * np.ones(n))

...
```

```
...
fig = plt.figure()
plt.plot(x, y, 'r.', markersize=12)
plt.plot(x, y_, 'g.-', markersize=12)
plt.plot(x, lr.predict(x[:, np.newaxis]), 'b-')
plt.gca().add_collection(lc)
plt.legend(('Data', 'Isotonic Fit', 'Linear Fit'), loc='lower right')
plt.title('Isotonic regression') plt.show()
```



Merci

Références :

- Stephen Marsland, *Machine Learning: An Algorithmic Perspective*, 2009
- http://fr.wikipedia.org/wiki/Python_%28langage%29
- http://scipy.org/
- http://scikit-learn.org/stable/
- http://docs.python.org/2/tutorial/

Remerciements :

- Sébastien Déjean
- Jérome Fehrenbach
- Vous

MERCI !!!