

INSTITUT
de MATHÉMATIQUES
de TOULOUSE

Python intermédiaire pour les scientifiques

Session 1/5 : Python général

Laurent Risser

Ingénieur de Recherche à l'Institut de Mathématiques de Toulouse

lrissier@math.univ-toulouse.fr

En bref :

1. Généralités sur Python (présentation, installation, utilisation, arguments de scripts).
2. Mini-TP 'hello world'.
3. Types de données (integer, float, string, listes, tuples, dictionnaires, compréhensions).
4. Structures de contrôle (if, while, for).
5. Fonctions.
6. Exceptions.
7. Utilisation de modules et de packages existants.
8. Présentation de Numpy (structure array, lecture/écriture de csv, copie par référence/valeur).
9. Présentation de matplotlib.
10. TP 'manipulation de python et de numpy/matplotlib'.

1) Généralités

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/
- Distributions « tout intégré » comme [enthought canopy](#) ou [anaconda](#) faciles à installer

Python 2 et 3 :

- Plusieurs *releases* de Python sont couramment utilisées : les 2.7 et 3.4
- Pour les sciences numériques, la 2.7 est la plus courante

1) Généralités

Utilisation de Python :

a) 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/Apple-clang-4.0.0.0.0)]
Type "help", "copyright", "credits" or "license()" for more
>>> 1+1
2
>>> █
```

```
blaye:tmp risser$ ipython
Python 2.7.2 (default, Oct 11 2012, 20:14:37)
Type "copyright", "credits" or "license()" for more

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

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

In [2]: █
```

b) 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
```

c) En tant que fichier compilé

→ Simplification de la distribution des exécutables

d) En tant que Notebook

→ Idéal pour communiquer sur des résultats et/ou pour la formation

1) Généralités

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

MyScript.py :

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

`python MyScript.py` (dans le shell)

→ Script name: MyScript.py
→ Number of arguments: 1
→ Arguments are: ['MyScript.py']

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

→ Script name: 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

1) Généralités

Quelques notes :

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

Note 2 : commentaires avec `#` ou bien `""" ... """`

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

Note 5 : Quitter Python avec *ctrl+d* ou bien `quit()`

3) Types de données

Basic data types : integer, float, boolean, string

Pas de déclaration de variables :

→ Taper `a=3` définit « a » comme un entier (integer) et lui donne la valeur 3

→ Taper `b=1.` définit « 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 3.4, le résultat est un float

Comparaison faite par `==`, `>`, `<`, `!=` (ou `<>`)

→ retourne un booléen True (1) ou False (0)

3) Types de données

Basic data types : integer, float, boolean, string

Informations sur une variable :

		<code>a=3</code>	
→ Valeur	:	<code>print(a)</code>	→ 3
→ Type	:	<code>type(a)</code>	→ int

Supprimer une variable (marche aussi pour les types à venir) :

```
a=3
a
→ 3

del(a)
a
-> NameError: name 'a' is not defined
```

3) Types de données

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
```

3) Types de données

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 1^{er} indice est 0 !)

```
Liste_A[1] → 3
```

Pour accéder 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] ← start:stop:step !!!  
liste_B[::-1] → [1, 6, 5, 4, 2, 3, 0]
```

3) Types de données

Basic data structures : List, Tuple, Dictionary

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([2,7,8,9]) → [1, 2, 3, 4, 'hi', 2, 7, 8, 9]
```

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

```
List.remove(2) → [1, 3, 4,'hi', 2, 7, 8, 9, [10, 11, 12]]
```

```
List.pop(2) → [1, 3, 'hi', 2, 7, 8, 9, [10, 11, 12]]
```

3) Types de données

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)`

3) Types de données

Basic data structures : List, Tuple, Dictionary

Tuples :

- Même chose qu'une liste mais ne peut 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
```

3) Types de données

Basic data structures : List, Tuple, Dictionary

Dictionnaires :

- 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)]
```

4) Structures de contrôle

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
```

5) Fonctions

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'

5) Fonctions

Il est possible de définir des valeur d'entrée par défaut:

```
def FunctionName(arg1, arg2, ... , optArg1=1, optArg2='g', ... ):  
    commands  
    return value
```

→ *Les arguments obligatoires sont en premier et dans un ordre pré-défini*

→ Les arguments optionnels peuvent être entré dans n'importe quel ordre

```
def pythagorus(x=1,y=1):  
    """ Computes the hypotenuse of two arguments """  
    r = pow(x**2+y**2,0.5)  
    return x,y,r  
  
pythagorus()      →      (1, 1, 1.4142135623730951)
```

6) Exceptions

Les exceptions sont un outil particulièrement efficace pour :

- Fiabiliser un code
- Débugger un code

```
for i in range(-2,2):  
    print(10./float(i))
```

→ -5.0

→ -10.0

→ Traceback (most recent call last):

File "<ipython-input-8-e9d805d2e8cf>", line 2, in <module>

```
    print(10./float(i))
```

ZeroDivisionError: float division by zero

6) Exceptions

Les exceptions sont un outil particulièrement efficace pour :

- Fiabiliser un code
- Débugger un code

```
for i in range(-2,2):  
    try:  
        print(10./float(i))  
    except:  
        print("A problem occured when dividing by"+str(i))
```

```
→ -5.0  
→ -10.0  
→ A problem occured when dividing by 0  
→ 10.0  
→ 5.0
```

→ Le code n'a pas planté

→ On a pu voir d'où venait l'erreur

6) Exceptions

On peut spécifier les exceptions pour différents types d'erreur :

```
for i in range(-2,2):
    try:
        print(10./float(i))
    except ZeroDivisionError:
        print("Zero division error captured for i="+str(i))
    except:
        print("An error was captured for i="+str(i))

→ -5.0
→ -10.0
→ Zero division error captured for i=0
→ ...
```

Une multitude d'erreurs sont prédéfinies dans Python :

StandardError, KeyboardInterrupt, OverflowError, EOFError, MemoryError, Warning, ...

(voir <https://docs.python.org/2.7/library/exceptions.html#builtin-exceptions>)

6) Exceptions

On peut de même choisir de remonter une exception dans un code :

```
raise NameError('Coucou')
```

→ Traceback (most recent call last):

→ File "<ipython-input-10-fed54a5e1f1a>", line 1, in <module>

→ raise NameError('Coucou')

→

→ NameError: Coucou

6) Exceptions

Remarque : Les Context managers possèdent gèrent automatiquement les erreurs. Le plus courant est `with`.

```
f=open('fichier.txt','w')
try:
    #operations sur f
except Exception:
    #on gere (ou non) l'erreur
finally:
    f.close()
```

Equivalent (sans la possibilité de gérer l'erreur)

```
with open('fichier.txt','w') as f:
    #operations sur f
```

7) Utilisation de modules et de packages existants

- 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

7) Utilisation de modules et de packages existants

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
```

Compilation du module :

- Lors de son 1^{er} 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')`

Sous modules :

- Certains gros modules contiennent des sous modules. On parle de *packages*.
- Pour charger un sous-module : `import scipy.stats` OU `from scipy import stats`

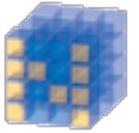
8) Numpy

Intérêt de Numpy, Matplotlib et Scipy :

→ Faire du calcul scientifique et de l'analyse de données sous Python

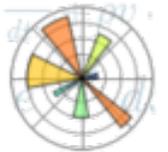
Numpy :

- 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 autres algorithmes numériques.
- Quelques fonctions redondantes avec celles de NumPy.



- Fonctions de SciPy plus évoluées 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

- De loin, la structure de donnée la plus utilisée pour les maths numériques sous Python
- Tableau en dimension $n = 1, 2, 3, \dots, 40$
- 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])
```



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.]])
```

- `linspace(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

- `a=np.array([[0,1],[2,3],[4,5]])`
- `np.ndim(a)` → 2 (Nombre de dimensions)
- `np.size(a)` → 6 (Nombre d'éléments)
- `np.shape(a)` → (3, 2) (Tuple contenant la dimension de *a*)
- `np.transpose(a)` → `array([[0, 2, 4],[1, 3, 5]])` (Transposé)
- `a.min()`, `np.min(a)` → 0 (Valeur min)
- `a.sum()`, `np.sum(a)` → 15 (Somme des valeurs)
- `a.sum(axis=0)` → `array([6, 9])` (Somme sur les colonnes)
- `a.sum(axis=1)` → `array([1, 5, 9])` (Somme sur les lignes)

mais aussi `max`, `mean`, `std`, ..., et encore :

- `np.concatenate([[1,2,3],[4,5,6]])` → `array([1,2,3,4,5,6])` (Concaténation de vecteurs)
- `np.concatenate([[1,2,3],[4,5,6]],axis=1)` → `array([[1,2,3,4,5,6]])` (Concaténation en ligne)
- `np.concatenate([[1,2,3],[4,5,6]],axis=0)` → `array([[1,2,3],[4,5,6]])` (Concaténation en colonne)
- `np.arange(6).reshape(3,2)` → `array([[0,1],[2,3],[4,5]])` (modification de la forme)

Fonctions compilées et optimisées → à utiliser en évitant les boucles `for` tant que possible



2.1.4) Copie d'un array

Comme pour les listes ...

... la copie d'un array fonctionne par référence !

Pour copier toutes les valeurs d'un array dans un autre, utilisez **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] )
```



2.1.5) Contrôler le type des Arrays

- 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')
```



2.1.6) Opérations de base sur les Arrays

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]])
```



2.1.7) Lecture/écriture d'un array à partir d'un fichier csv

Il existe une *basic data structure* File :

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

... mais pour les arrays, 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 : On verra dans la session 5 d'autres outils de lecture de données



2.1.8) Fonction lambda sur un array

→ 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.])
```

9) Matplotlib

→ Ressemble aux fonctions de graphes 2D de Matlab...

→ On appelle Matplotlib avec : `import matplotlib.pyplot`

en mieux 😊

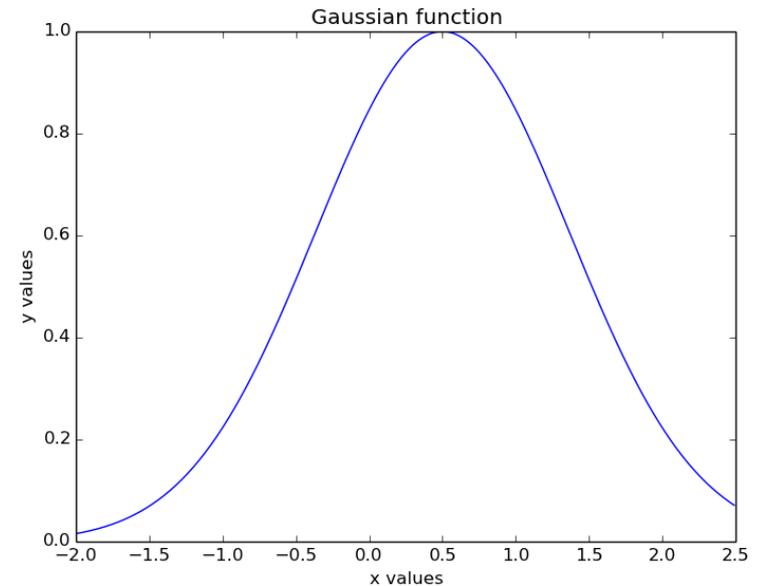


```
import numpy as np
import matplotlib.pyplot as plt

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

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

plt.clf() #en option
```



Remarque :

→ Il existe aussi pylab qui a été écrit pour ressembler plus à Matlab. Il n'est plus recommandé.

9) Matplotlib

→ Sauvegarder une image dans un fichier



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

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

plt.plot(x,y)
plt.xlabel('x values')
plt.ylabel('y values')
plt.title('Gaussian function')
plt.savefig('toto.pdf')

plt.clf()
```

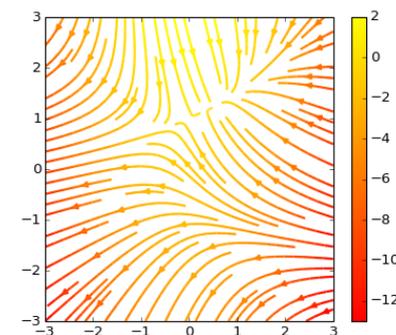
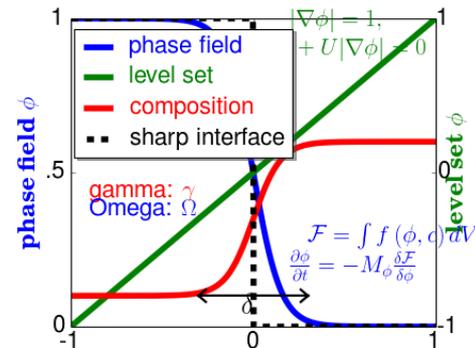
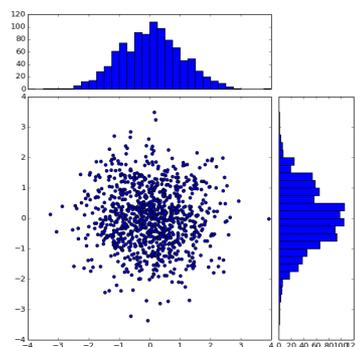
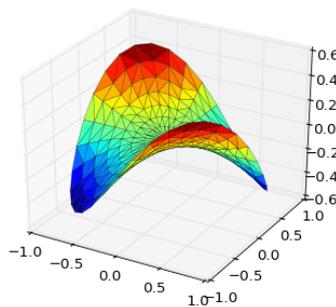
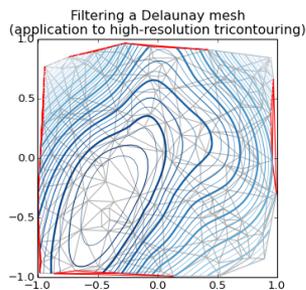
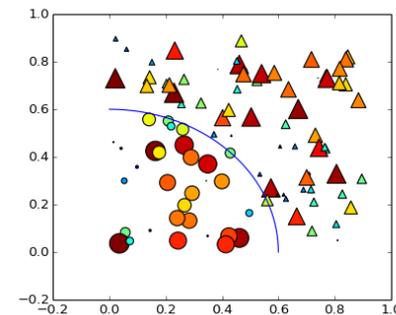
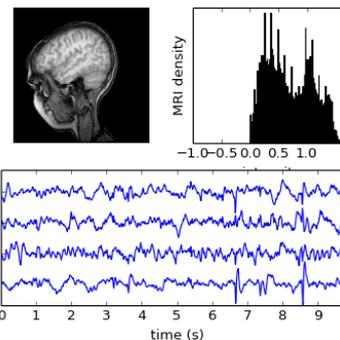
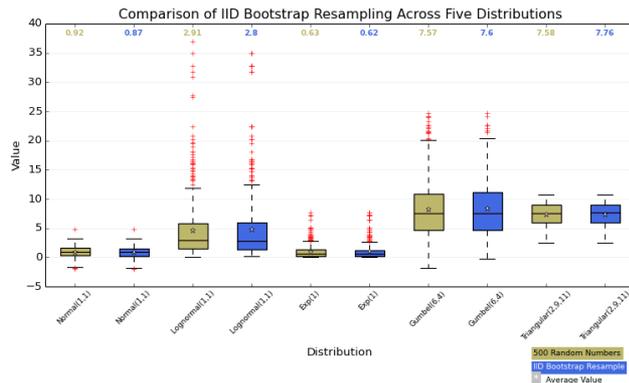
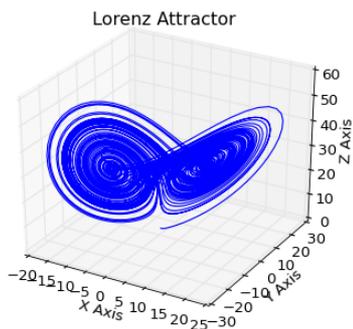


Fichier toto.pdf

9) Matplotlib

Pour se faire une idée de ce que l'on peut faire avec MatplotLib:

<http://matplotlib.org/gallery.html>



...

→ On trouve le code python correspondant à chaque figure sur la page

Références :

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

Remerciements :

- Etienne Gondet
- Sébastien Déjean
- Jérôme Fehrenbach
- Vous

MERCI !!!