INSTITUT
de MATHEMATIQUES
de TOULOUSE

T7.AP01 : Initiation à Python

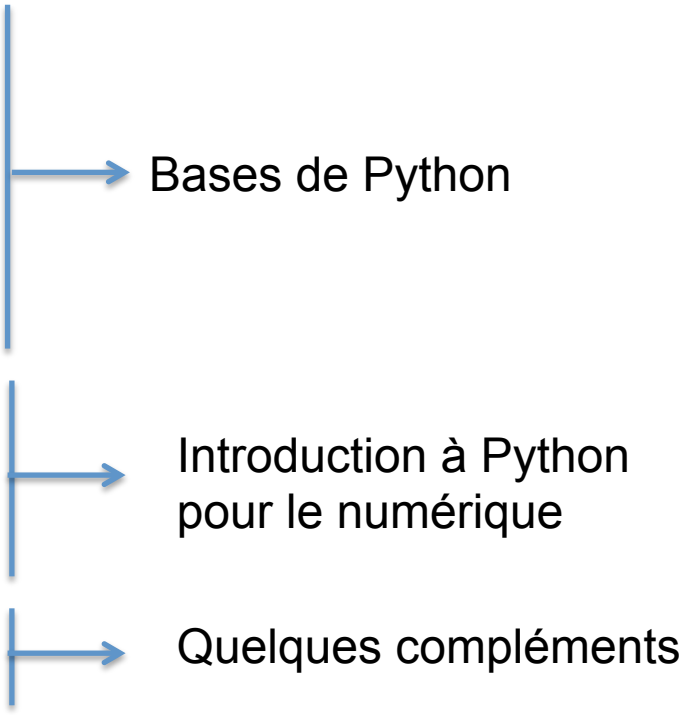
Laurent Risser
Ingénieur de Recherche à l'Institut de Mathématiques de Toulouse

lrisser@math.univ-toulouse.fr

Résumé

Plan:

1. Généralités sur Python.
2. Types de données.
3. Structures de contrôle.
4. Fonctions.
5. Exceptions.
6. Utilisation de modules et de packages existants.
7. Numpy.
8. Matplotlib et SciPy
9. Lecture de fichiers de données
10. Portabilité
11. Création de Notebooks



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)

Une des grandes forces de Python est la taille de sa communauté

Installation :

- Voir www.python.org/getit/
- Distributions « tout intégré » comme anaconda ou enthought canopy faciles à installer

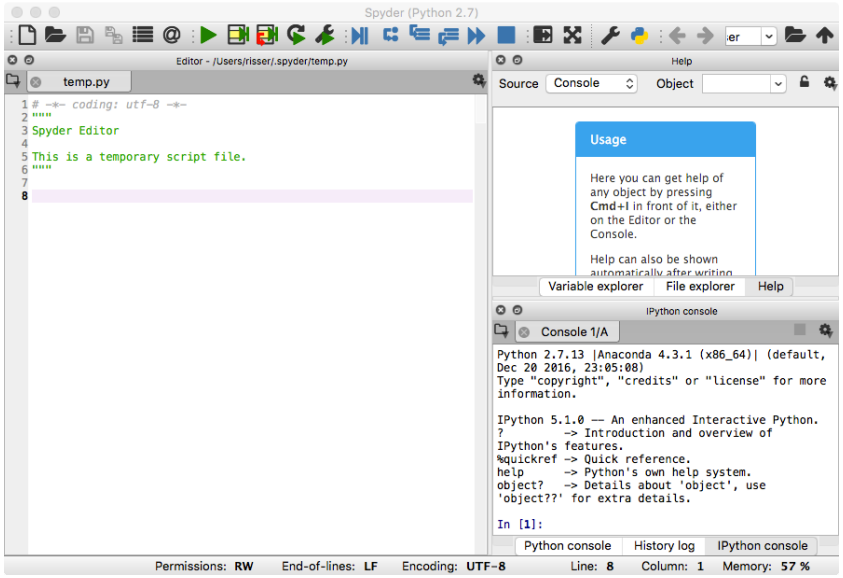
Python 2 et 3 :

- Plusieurs *releases* de Python sont couramment utilisées : les 2.7 et 3.6
- Pour les sciences numériques, la 2.7 est la plus courante

Utilisation de Python :

a) Sous IDLE, ipython ou bien Spyder

```
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]:
```



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 Notebook (jupyter ou ipython)

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

Quelques notes :

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

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

Note 3 : Quitter Python avec *ctrl+d* ou bien `quit()`

2) Types de données

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 3.4, le résultat est un float

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

Basic data types : integer, float, boolean, string

Informations sur une variable :

→ Valeur :
→ Type :

```
a=3
print(a)    → 3
type(a)     → int
```

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

```
a=3
a
→ 3

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

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

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]]
```

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)`

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)]
```

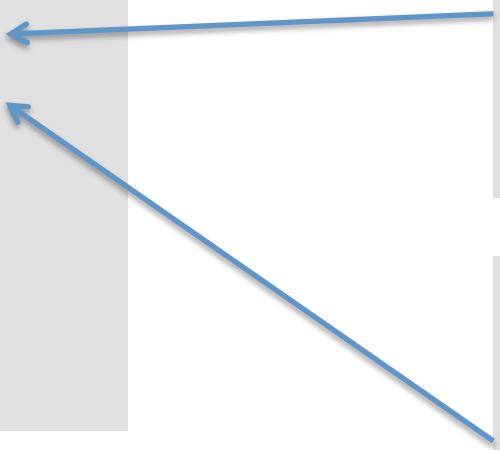
Les blocs de codes sont définis par deux points suivi d'une indentation fixe.
→ intérêt : forcer à écrire du code facile à lire

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'
```

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)
```

5) 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
```

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

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#bltin-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
```

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
```

6) 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

6) 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 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 :

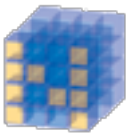
- Listés dans `sys.path` (après un `import sys`)
- Ajout avec `sys.path.append('mypath')`

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`

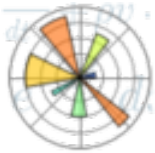
Intérêt de Numpy, Matplotlib et Scipy :
→ Faire du calcul scientifique et de l'analyse de données sous Python

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 :



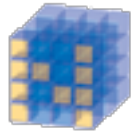
- Fonctions de visualisation / graphs
- Commandes proches de celles sous matlab (mais un résultat plus sympa)
- Aussi connu sous le nom de pylab

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

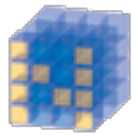


NumPy

- 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

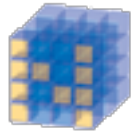
```
m=np.mgrid[0:3,0:2]
m → array([ [[0, 0], [1, 1], [2, 2]] ,
            [[0, 1], [0, 1], [0, 1]] ])
```

```
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

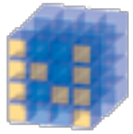
- a=np.array([[0,1],[2,3],[4,5]])
- np.ndim(a) → 2 (Nombre de dimensions)
- np.size(a) → 6 (Nombre d'éléments)
- np.shape(a) → (3, 2) (Tuple contenant la dimension de *a*)
- np.transpose(a) → array([[0, 2, 4],[1, 3, 5]]) (Transposé)
- a.min(), np.min(a) → 0 (Valeur min)
- a.sum(), np.sum(a) → 15 (Somme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



NumPy

Comme pour les listes …

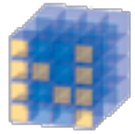
… la copie d'un array fonctionne par référence !

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



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])
```

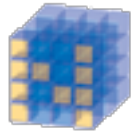
Convertir le type d'un array :

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

Connaitre le type d'un array :

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

2.1.6) 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]])
```

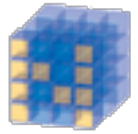
```
c=np.transpose(b)
→     array([[3, 5, 7],
             [4, 6, 8]])
```

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]])
```

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



NumPy

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 :

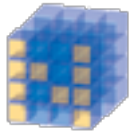
1 , 2 , 3
4 , 5 , 6
7 , 8 , 9

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



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)
```

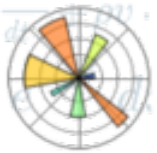
```
x    →    array([-2.    , -1.5   , -1.    , -0.5    ,  0. ,  0.5    , 1.    ])
y    →    array([0.0694,  0.2231,  0.5134 ,  0.8464,  1.  ,  0.8464,  0.5134])
```

```
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)
```

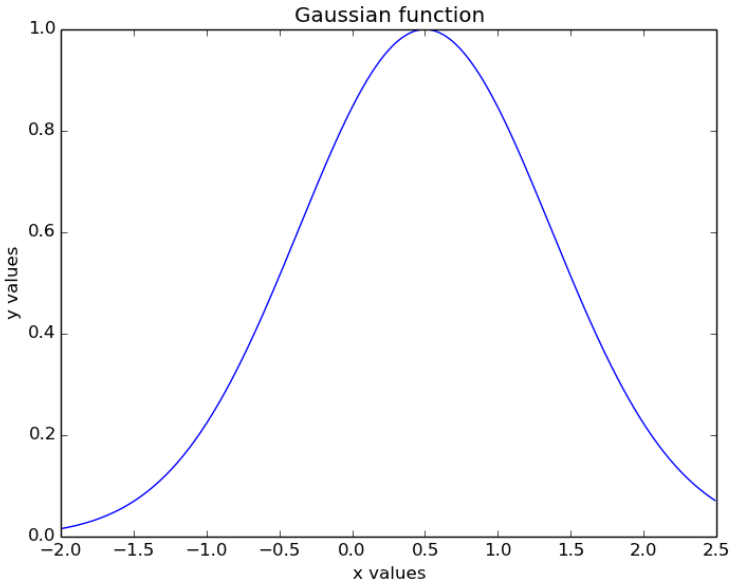
```
a    →    array([ 0. ,  0.5,  1. ,  1.5])
b    →    array([ 10. ,  10.5,  11. ,  11.5])
c    →    array([ 20.,  22.,  24.,  26.])
```

→ Ressemble aux fonctions de graphs 2D de Matlab… en mieux

→ On appel la Matplotlib avec : `import matplotlib.pyplot`



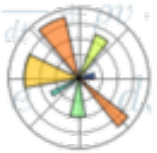

```
import numpy as np
import matplotlib.pyplot as plt

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.

→ Sauvegarder une image dans un fichier

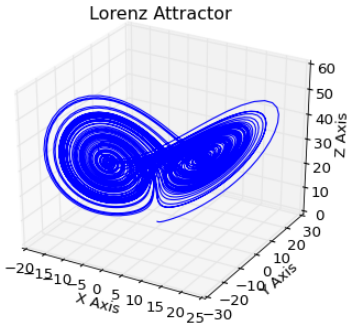


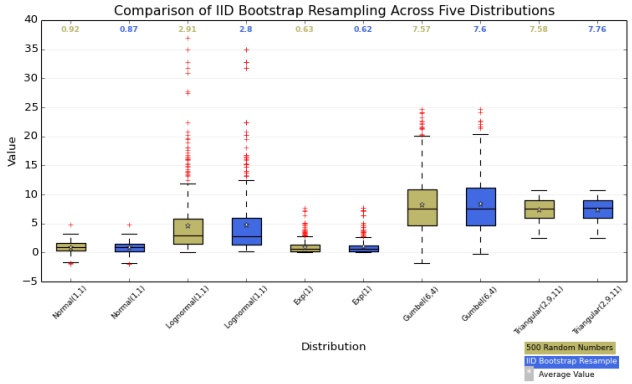

```
import numpy as np
import matplotlib.pyplot as plt

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()
```

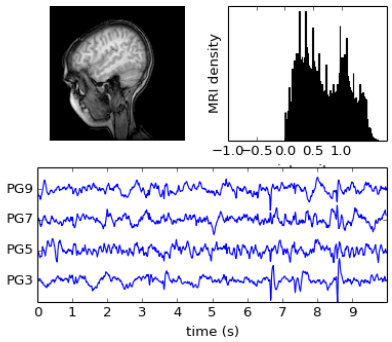
Fichier toto.pdf

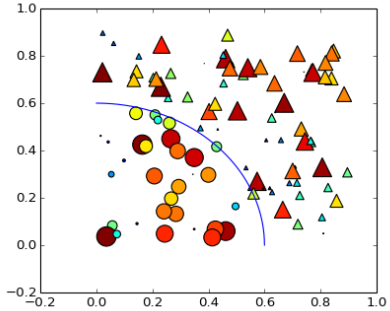
Pour se faire une idée de ce que l'on peut faire avec MatplotLib:
http://matplotlib.org/gallery.html

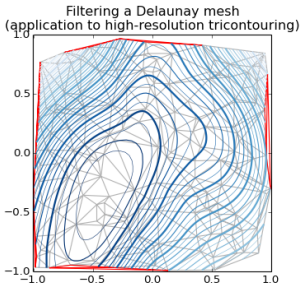
Matplotlib

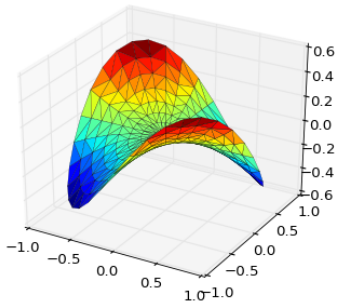


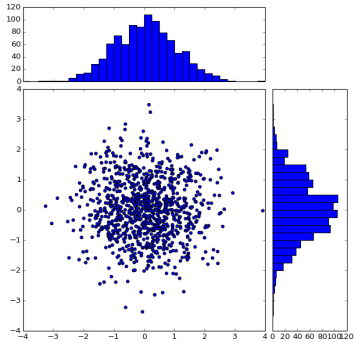


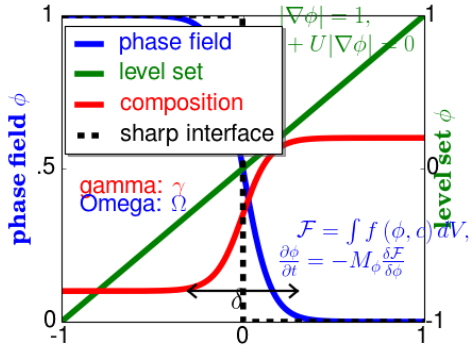


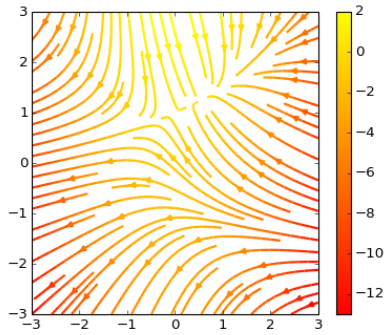












…

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

Scipy est une bibliothèque pour les mathématiques, la science, et l'ingénierie

→ Utilise massivement la classe 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)
- … et bien d'autres encore

Technique de base pour les fichiers ascii : Utilisation de l'objet file

Fichier testFile.csv :

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

```
f = open('test.csv', 'r')

toto=f.read()
print toto

→ '1,2,3\n4,5,6\n7,8,9\n'

f.close()
```

Technique de base pour les fichiers ascii : Utilisation de l'objet file

Fichier testFile.csv :

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

```
f = open('test.csv', 'r')
lines=f.readlines()

for i in range(len(lines)):
        print lines[i]

→ 1,2,3
→ 4,5,6
→ 7,8,9


f.close()
```

Technique de base pour les fichiers ascii : Utilisation de l'objet file

Fichier testFile.csv :

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

```
f = open('test.csv', 'r')
lines=f.readlines()


for i in range(len(lines)):
        values=lines[i].split(',')
        print values[0]+' et '+values[1]

→ 1 et 3
→ 4 et 6
→ 7 et 9

f.close()
```

Remarque : Les values sont des *string* ici. Les convertir en *float* pour les traiter

Technique de base pour les fichiers ascii ou **binaires** :

Fichier testFile.csv :

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

```
f = open('test.csv', 'r')
f.seek(2)
print f.read(1)    →    2

f.seek(5)
f.read(1)=='\n'    →    True

f.seek(6)
a=f.read(3)        →    4,5
print a

f.seek(-2,2)
print f.read(1)    →    4,5

f.close()
```

Technique de base pour les fichiers ascii ou **binaires** :

Fichier testFile.csv :

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

```
f = open('test.csv', 'r')
f.seek(2)
print f.read(1)      →    2

f.seek(5)
f.read(1)=='\n'      →    True

f.seek(6)
a=f.read(3)          →    4,5
print a

f.seek(-2,2)
print f.read(1)      →    9

f.close()
```

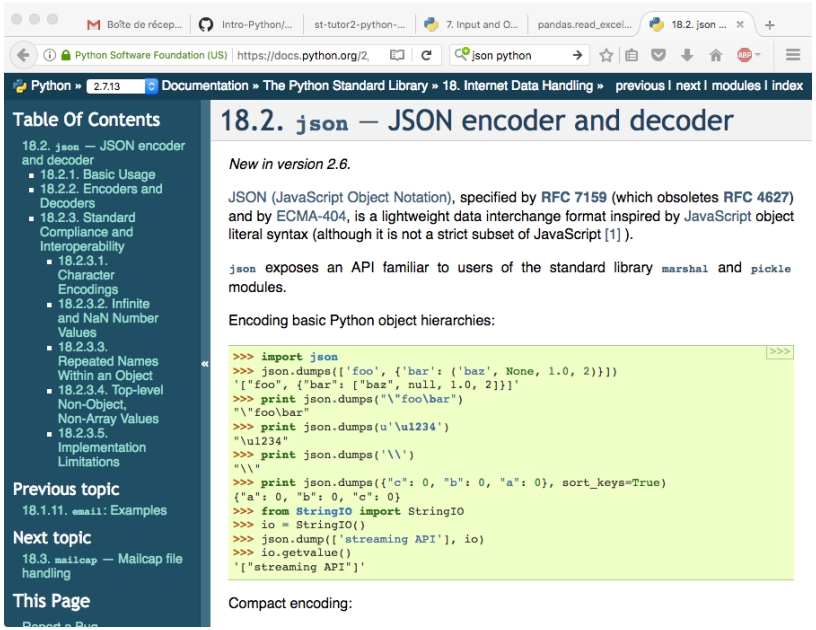
Localisation initiale :
0 : Début fichier
1 : localisation courante
2 : Fin fichier

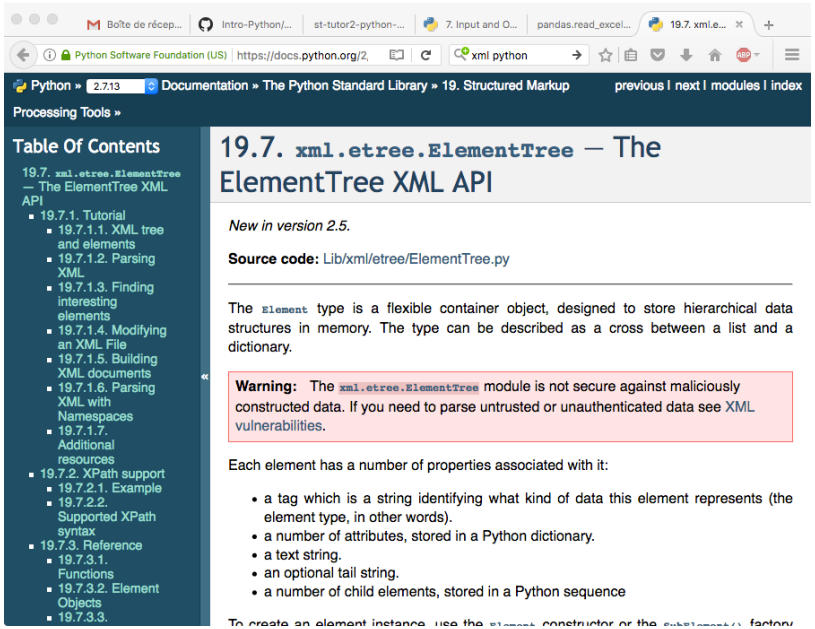
Chemin à parcourir en octets

Et pour des fichiers avec une structure ascii plus complexe comme les xml ou json ???

```
import json

…
```

```
import xml.etree.ElementTree as ET

…
```





→ https://docs.python.org/2/library/json.html
→ https://docs.python.org/2/library/xml.etree.elementtree.html
→ et beaucoup d'autres plus spécialisés … demandez à google ☺

Et pour des fichiers binaires Matlab ou Excel ???

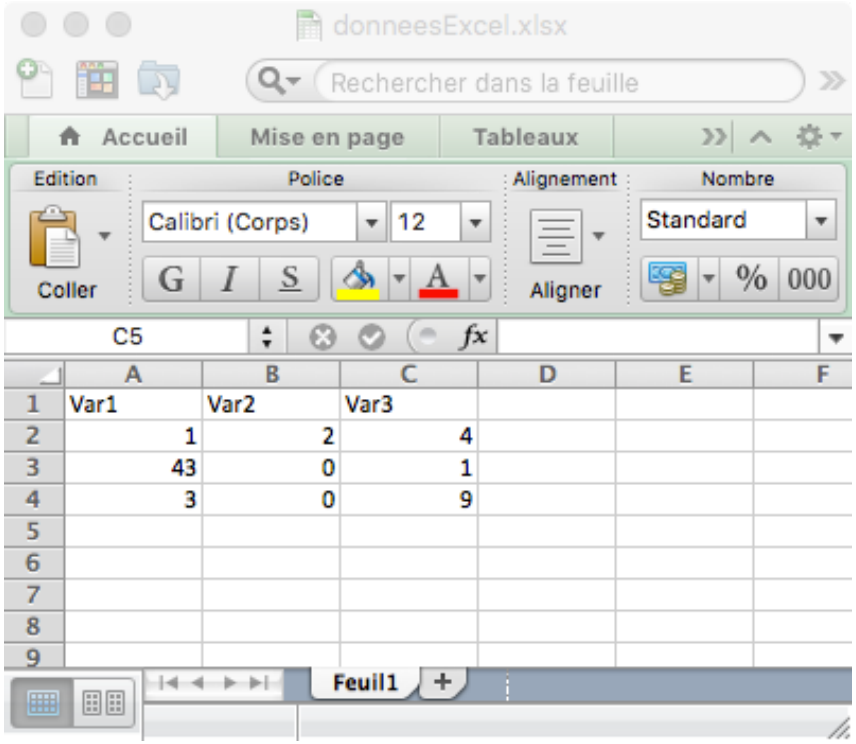
Matlab :

```
import scipy.io as sio
test = sio.loadmat('test.mat')
```

Excel :

```
from pandas import read_excel

myframe=read_excel('ExcelData.xlsx')

print myframe
```

```
→      Var1   Var2   Var3
→  0      1      2      4
→  1     43      0      1
→  2      3      0      9
```



A de rares exceptions près tout code Python est portable Windows/MAC/Linux.

Toutefois, pour tester le système, utiliser sys.platform Pour les systèmes les plus courants, les valeurs sont :

Linux	→	'linux2' (ou 'linux' après Python 3.3)
Windows	→	'win32'
Windows/Cygwin	→	'cygwin'
Mac OS X	→	'darwin'

Pour quasi tous les codes *scientifiques*, les noms de répertoires sont le seul problème.
→ Utiliser : os.path.join()

Sous Linux ou MacOS :

```
import os
SvgFile=os.path.join(".","DirOutputs","resultats.csv")
print SvgFile
→ ./DirOutputs/resultats.csv
```

Sous Windows :

```
import os
SvgFile=os.path.join(".","DirOutputs","resultats.csv")
print SvgFile
→ .\DirOutputs\resultats.csv
```

Python 2.7 reste le plus utilisé dans un cadre scientifique mais devrait ne plus être supporté un jour ou l'autre.

→ la date d'arrêt du support est prévu en 2020 mais la date *pourrait* être repoussée

→ Python 3 assure la rétrocompatibilité de la grande majorité des fonctions Python 2.7

Principales différences :

PYTHON 2.7	→	PYTHON 3	
`print "Bonjour"`	→	`print("Bonjour")`	
`print "Val=", 3`	→	`print("Val", 3)`	
`raise IOError, "file error"`	→	`raise IOError("file error")`	-- (appel des exceptions)
`Tkinter`	→	`tkinter`	-- (nom de quelques modules)
`xrange()`	→	`range()`	-- (nom de quelques objets)

Remarque :

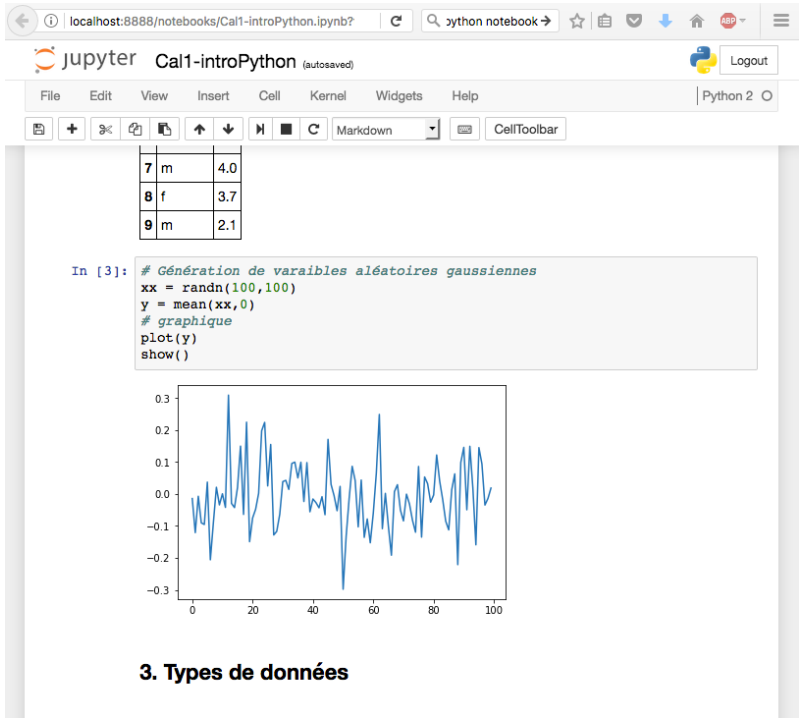
`print("Bonjour")`, `raise IOError("file error")`, `range()` , … fonctionnent sous les versions récentes de 2.7.

→ Autant les utiliser dès maintenant.

11) Création de notebooks

Jupyter Notebooks (anciennement IPython Notebook) :

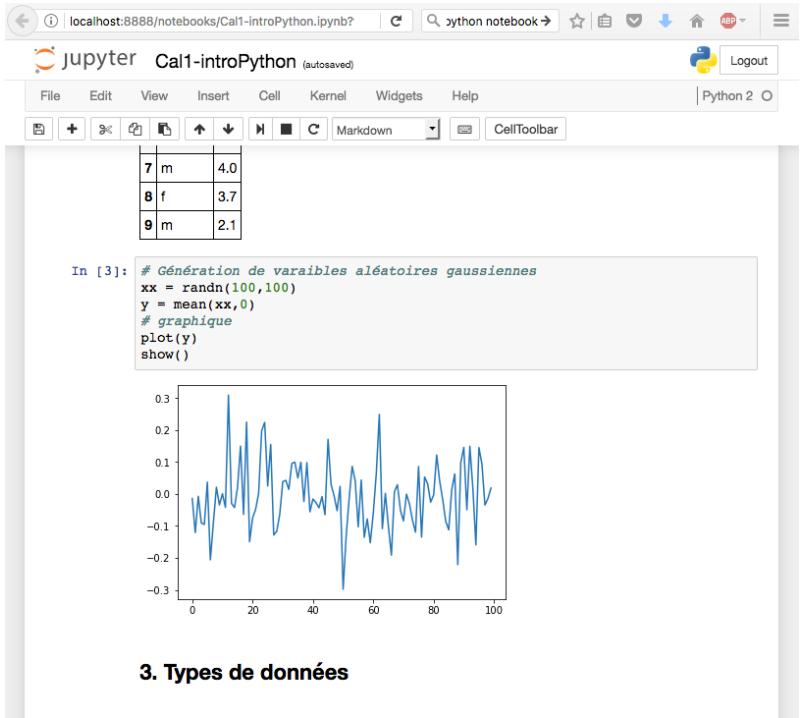
- Environnement *interactif* qui peut combiner du texte enrichi, l'exécution de code python, et des graphiques.
- Idéal pour la communication de résultats.
- Idéal pour l'enseignement des sciences numérique par la pratique.
- Intégré dans Anaconda et dans iPython



11) Création de notebooks

Jupyter Notebooks (anciennement IPython Notebook) :

- Environnement *interactif* qui peut combiner du texte enrichi, l'exécution de code python, et des graphiques.
- Idéal pour la communication de résultats.
- Idéal pour l'enseignement des sciences numérique par la pratique.
- Intégré dans Anaconda et dans iPython



Lecture d'un notebook à partir d'un fichier toto.ipynb :

→ Dans un terminal taper la commande

```
jupyter notebook toto.ipynb
```

ou bien :

```
ipython notebook toto.ipynb
```

→ Le fichier s'ouvre dans le navigateur internet

Anaconda navigator (si installé) peut aussi ouvrir le fichier toto.ipynb via l'explorateur Jupyter

11) Création de notebooks

Jupyter Notebooks (anciennement IPython Notebook) :

- Environnement *interactif* qui peut combiner du texte enrichi, l'exécution de code python, et des graphiques.
- Idéal pour la communication de résultats.
- Idéal pour l'enseignement des sciences numérique par la pratique.
- Intégré dans Anaconda et dans iPython

Créer un nouveau notebook :

→ Dans un terminal taper la commande

```
jupyter notebook
```

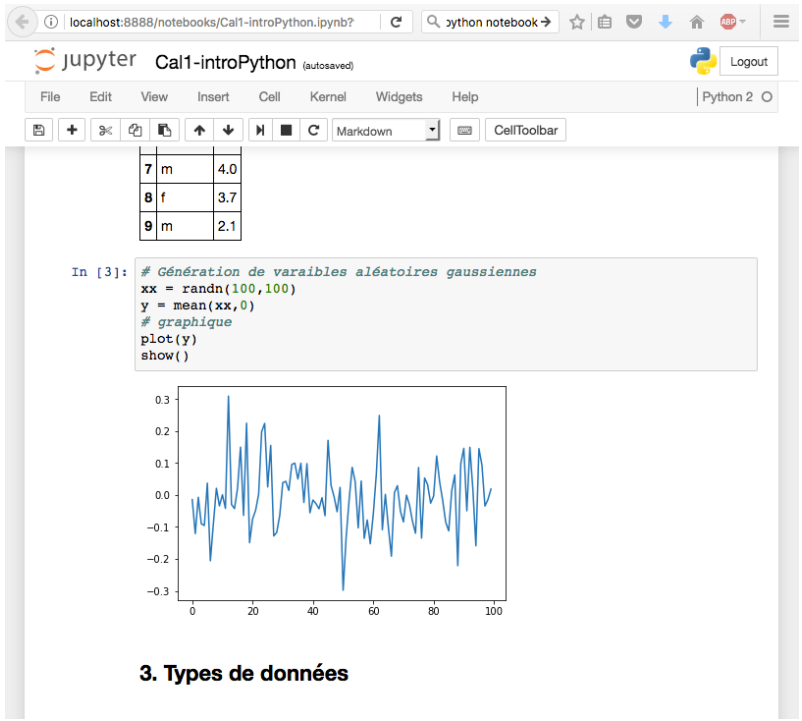
ou bien :

```
ipython notebook
```

→ Aller dans *New* et sélectionner *Python 2*

→ Le code s'écrit en mode *Code* et le texte en mode *Markdown*

→ On créé de nouvelles zones avec les *Cells*



Merci

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
- Jérome Fehrenbach
- Vous

MERCI !!!