Introduction au langage Python

Emmanuel Marcq, Loïc Rossi

LATMOS, Université de Versailles St-Quentin-en-Yvelines

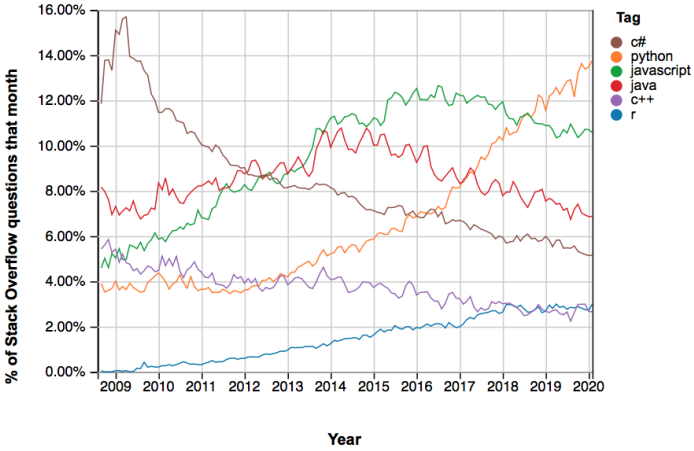
M2 Planétologie Île-de-France – 2020/2021

- Python est un langage **interprété**
  (par opposition à **compilé** comme C ou Fortran).

  Avantages syntaxe plus légère, développement plus rapide
  Inconvénients lenteur d'exécution, **mais** possibilité de précompiler certaines bibliothèques.

- Python est **libre** et **gratuit**
  (contrairement à Matlab, IDL, etc.)

- Python est un langage **généraliste** : nombreuses bibliothèques dans tous les domaines.
  - Web, base de données, *machine learning*, etc.



Rank	Language	Type	Score
1	Python		100.0
2	Java		96.3
3	C		94.4
4	C++		87.5
5	R		81.5
6	JavaScript		79.4
7	C#		74.5
8	Matlab		70.6
9	Swift		69.1
10	Go		68.0

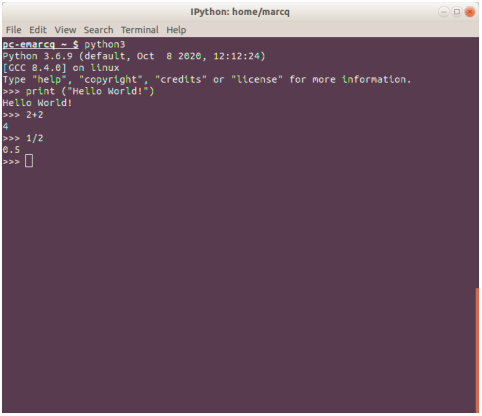
Classement IEEE (ingénieurs)

Remarque importante

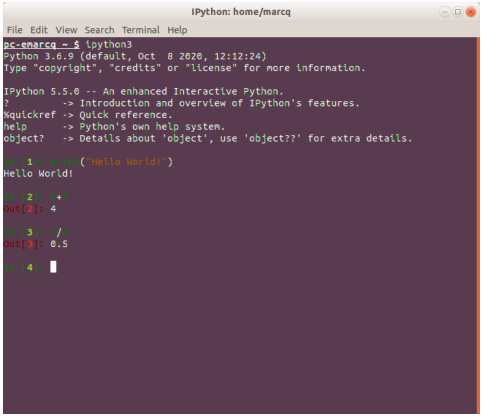
- La version actuelle de Python est la version 3.9 (Octobre 2020)
- Le passage de la version 2 à la version 3 est significatif : **un code écrit pour Python 3 n'est pas compatiable avec Python version 2.7** (la dernière version de Python 2, qui n'est plus maintenue depuis janvier 2020).

$\Rightarrow$ Mieux vaut écrire les nouveaux codes en Python 3, mais Python 2.7 est encore très répandu : attention au contexte informatique !

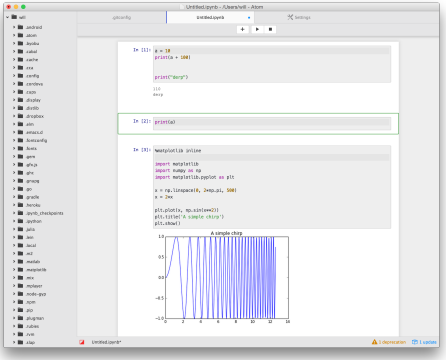
- Le plus simple : écrire directement le code dans l'interpréteur Python standard



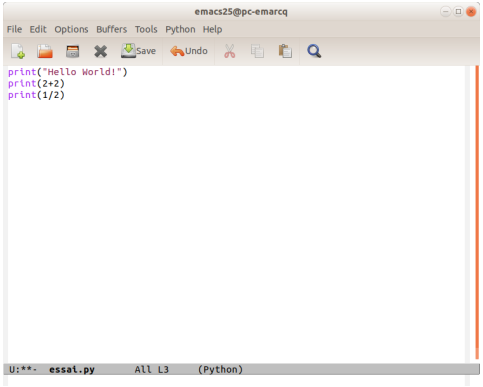
- Le plus simple : écrire directement le code dans l'interpréteur Python standard
- Mieux : utiliser un interpréteur augmenté comme *IPython*



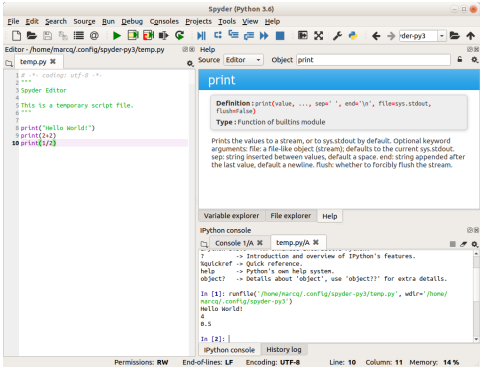
- Le plus simple : écrire directement le code dans l'interpréteur Python standard
- Mieux : utiliser un interpréteur augmenté comme *IPython*
- Ou utiliser un *notebook* Jupyter



- Le plus simple : écrire directement le code dans l'interpréteur Python standard
- Mieux : utiliser un interpréteur augmenté comme *IPython*
- Ou utiliser un *notebook* Jupyter
- Pour des programmes plus complexes que quelques lignes : écrire le code dans un éditeur de texte
  - code source fichier texte, d'extension usuelle `*.py`
  - exécution avec `python fichier.py`



- Le plus simple : écrire directement le code dans l'interpréteur Python standard
- Mieux : utiliser un interpréteur augmenté comme *IPython*
- Ou utiliser un *notebook* Jupyter
- Pour des programmes plus complexes que quelques lignes : écrire le code dans un éditeur de texte
  - code source fichier texte, d'extension usuelle `*.py`
  - exécution avec `python fichier.py`
- Ou dans un environnement de développement (IDE) tel que *Spyder*



Exemple de code

```
# Ceci est un commentaire
print('Hello '+"World")
for i in [1,2,3]:
    j = i*2 # on double
    print('i=',i,' et j=',j)
    if i==2:
        print('Ici i=2')
```

Résultat

```
Hello World
i= 1 et j= 2
i= 2 et j= 4
Ici i=2
i= 3 et j= 6
```

- Tout ce qui suit # est ignoré (commentaires)
- Les chaînes de caractères sont délimitées indifféremment par deux ' ou deux "
- Les blocs commencent après le caractère « : » et sont délimités par **l'indentation** (4 espaces recommandés)
- Pas de déclaration des variables
- Pas de caractère de fin de ligne
- Python est sensible à la **casse** (majuscules/minuscules)

```
>>> x = 3
>>> y = 7.
>>> z = x+y*1j
>>> t = x**y # exponentiation
>>> print(z)
(3+7j)
>>> print(t)
2187.0
>>> print(type(x),type(y))
<class 'int'> <class 'float'>
>>> print(type(z))
<class 'complex'>
>>> "Hello "+"World"
'Hello World'
```

- Booléens `True`, `False`
- Entiers taille arbitraire
- Flottants de $\sim 10^{-307}$ à $\sim 10^{307}$
- Complexes
- Chaînes de caractères

Attention !

En Python 2, la division de deux entiers renvoie un nombre entier !

```
>>> print(1/2) # 1 entier
>>> 0
>>> print(1./2) # 1 flottant
>>> 0.5
```

```
>>> L1 = [1, 2, 3]
>>> L2 = ['oui', 'non', L1]
>>> L2[0]
'oui'
>>> L2[2][1]
2
>>> ch = 'python'
>>> ch[0]
'p'
>>> dico = {'nom':"Einstein",'age':42}
>>> dico['nom']
'Einstein'
>>> dico['age']
42
```

Listes délimitées par [et]

- Accès aux éléments via leur **indice**
- Peut contenir des éléments de n'importe quel type (y compris d'autres conteneurs)
- Une chaîne de caractères est aussi une liste de ses caractères individuels

Tuples délimités par (et)

- $\approx$ liste en lecture seule
- Surtout utilisés en entrée/sortie de fonctions.

Dictionnaires délimités par { et }

- Accès aux éléments via leur **clé**

```
>>> x = [4, -3, -5, 6, 12,
... -7, 8, 9, 10]
>>> x[1]
-3
>>> x[-1]
10
>>> x[0:5]
[4, -3, -5, 6, 12]
>>> x[5:]
[-7, 8, 9, 10]
>>> x[3:10:2]
[6, -7, 9]
>>> x[::-1]
[10, 9, 8, -7, 12, 6, -5, -3, 4]
```

- Une liste est un ensemble ordonné à une dimension
- Le premier indice est 0 comme en C
  - $\neq$ Matlab ou Fortran !
- Les indices négatifs comptent à partir de la fin
- Les intervalles se notent `debut:fin`
  - `debut` inclus, `fin` exclu
  - Les indices «évidents» peuvent être omis
  - On peut préciser l'incrément `debut:fin:increment`
  - $\neq$ Matlab (`debut:increment:fin`) !

Copie de conteneurs

Attention !

Une copie brute pointe vers le même objet en mémoire :

```
>>> A = [1, 2, 3]
>>> B = A
>>> A[2] = 8
>>> A
[1, 2, 8]
>>> B
[1, 2, 8]
```

Pour éviter cela, on copie plutôt les **éléments** :

```
>>> A = [1, 2, 3]
>>> B = A[:]
>>> A[2] = 8
>>> A
[1, 2, 8]
>>> B
[1, 2, 3]
```

Opérateurs arithmétiques

Addition, soustraction, multiplication, division, exponentiation, modulo, division entière : `+` `-` `*` `/` `**` `%` `//`

Opérateur de concaténation pour chaînes de caractères (ou listes)

`[1,2]+[2,3]` → `[1,2,2,3]`
`'bon'+'jour'` → `'bonjour'`

Opérateurs logiques

Renvoient un booléen `True` ou `False`

- Comparaisons `<` `>` `<=` `>=` `==` `!=`
- Connecteurs `and` `or` `not`
- Appartenance `in`
- Identité `is`

```
>>> listex = [0,2,3,4]
>>> for x in listex:
...     print('x*2 =',x*2)
...
x*2 = 0
x*2 = 4
x*2 = 6
x*2 = 8
>>> fichiers = ['a.txt','b.txt']
>>> for fich in fichiers:
...     print(fich)
...
a.txt
b.txt
```

- Elle généralement suivies du caractère : et d'un bloc **indenté**

  boucle `for`
  - répète le bloc pour toutes les valeurs de la variable dans la liste
  - la variable d'itération n'est pas forcément numérique !

```
>>> x = 0
>>> while x<3:
...     x = x+1
...     print(x)
1
2
3
>>> x = 4
>>> if x<3:
...     print('x petit')
... elif x<6:
...     print('x moyen')
... else:
...     print('x grand')
x moyen
```

boucle `while`

- répète le bloc tant que la condition est vérifiée.
- en particulier, `while True:` commence une boucle infinie.

test `if/elif/else`

- Si la condition après `if` est vérifiée, le bloc suivant est exécuté.
- Sinon (et seulement sinon), la condition suivant le prochain `elif` est testée aussi (et le bloc exécuté le cas échéant).
- Si aucune condition n'est vérifiée, le bloc suivant `else` est exécuté.

try...except

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

La gestion des erreurs (appelées **exceptions** en Python) prévisibles se fait de la façon suivante :

```
>>> try:
...     print(1/0)
... except ZeroDivisionError:
...     print('Division par 0!')
Division par 0!
```

On peut aussi utiliser except sans code d'erreur, auquel cas toutes les erreurs sont récupérées par le bloc suivant. **À n'utiliser qu'avec précaution !**

Définition : exemple

```
>>> def mult(a,b,plus_un=False):
...     prod = a * b
...     if plus_un is True:
...       prod = prod + 1
...     return prod
```

Ici, `a` et `b` sont les **arguments** obligatoires de la fonction `mult`, et `plus_un` est un argument optionnel (valant `False` par défaut).
`prod` est alors la variable **retournée** par la fonction `mult`. Une fonction peut aussi renvoyer plusieurs variables sous forme d'un tuple.

Utilisation

```
>>> mult(3,4)
12
```

```
>>> mult(3,5,plus_un=True)
16
```

```
>>> class Etoile:
...     nom = 'Soleil'
...     def get_T(self):
...         return 5777.
>>> A = Etoile()
>>> A.nom
'Soleil'
>>> A.get_T()
5777.0
>>> A.nom = 'Sirius'
>>> A.nom
'Sirius'
```

- Notion centrale pour la programmation **orientée objet**
- Permet de créer des **structures de données** plus complexes que celles disponibles par défaut
- On accède aux **méthodes** (fonctions) ou aux **attributs** (variables) d'une **instance** d'une classe donnée en suivant la syntaxe `instance.methode()` ou `instance.attribut`
- Possibilités extrêmement riches : surcharge d'opérateurs, fonctions d'affichage, fonctions d'initialisation, etc.

monmodule.py

```
x = 42.
nom = "coucou"
```

```
>>> import monmodule
>>> print(monmodule.nom)
coucou
>>> import monmodule as m
>>> print(m.x)
42.
>>> from monmodule import nom
>>> print(nom)
coucou
```

- Un **module** est un ensemble de fonctions, variables, classes, autres modules, etc. spécialisés et disponibles dans un fichier séparé.
- On peut **importer** tout ou partie du contenu du module pour l'utiliser dans son propre code
- Intérêt majeur : préserver l'**espace de noms** des collisions possibles.

Pourquoi NumPy ?

Le type «liste» de Python est inadapté en sciences :

- une seule dimension d'indexation ;
- opérations arithmétiques contreintuitives.

NumPy introduit un nouveau type de données, le **tableau** (`array`), qui corrige ces défauts.

```
>>> x = [0,1,2]
>>> print(x*2)
[0, 1, 2, 0, 1, 2]
```

```
>>> import numpy as np
>>> x = np.array([0,1,2])
>>> print(x*2)
[0 2 4]
```

Création de tableaux

```
>>> Z = np.asarray([1,2,3])
>>> Z = np.zeros(6)                    >>> Z = np.ones(5)
>>> print(Z)                           >>> print(Z)
[0. 0. 0. 0. 0. 0. ]                   [1. 1. 1. 1. 1. ]

>>> L = np.linspace(0,5,9) # début, fin, nombre de valeurs
>>> A = np.arange(0,5,1.1) # début, fin, pas
>>> print(L)
[0.    0.625 1.25  1.875 2.5   3.125 3.75  4.375 5.   ]
>>> print(A)
[0.  1.1 2.2 3.3 4.4]
```

Tableaux à plusieurs dimensions

```
>>> T = np.zeros((3,7))
>>> print(T.shape) # affiche les dimensions de T
(3, 7)
>>> print(T)
[[0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]]
>>> T[2,:] = 4. # 1er indice = ligne
>>> T[0,-1] = 12
>>> print(T)
[[0. 0. 0. 0. 0. 0. 12.]
 [0. 0. 0. 0. 0. 0.  0.]
 [4. 4. 4. 4. 4. 4.  4.]]
```

Masques logiques

```
>>> X = np.linspace(0.,2.,5)
>>> Y = X**2
>>> mask = X > 1
>>> print(X)
[0. 0.5 1. 1.5 2. ]
>>> print(mask)
[False False False True True]
>>> print(X[mask])
[1.5 2. ]
>>> print Y[mask]
[2.25 4. ]
```

- Une opération logique appliqué à un tableau revoie un **masque** booléen.
- Ce masque en indice d'un tableau (de mêmes dimensions) sélectionne les seuls éléments pour lesquels l'opération logique est vérifiée.

Méthodes

```
>>> A = np.array([[1,4],[9,16]])
>>> print(A)
[[1  4]
 [9 16]]
>>> print(A.mean())
7.5
>>> print(A.std())
5.678908345800274
>>> print(A.min(),A.max())
1 16
>>> print(A.min(axis=0))
[1 4]
```

```
>>> print(A.shape)
(2, 2)
>>> print(A.T)
[[1  9]
 [4 16]]
>>> print(A.reshape(4,1))
[[ 1]
 [ 4]
 [ 9]
 [16]]
```

Fonctions

fichier.txt

```
  0.1   1.2   2.3
  3.    4.    5.
  12.   14.   17.
```

```
>>> A = np.loadtxt('fichier.txt')
>>> print(A)
[[0.1  1.2  2.3]
 [ 3.   4.   5.]
 [12.  14.  17.]]
```

Trigonométrie `np.sin`, `np.cos`, `np.tan`, etc.

Statistiques `np.mean`, `np.max`, `np.std`, `np.median`, etc.

Algèbre linéaire `np.matmul`, `np.linalg.solve`, etc.

Interpolation `np.interp`

Entrées/Sorties `np.load`, `np.save`, `np.loadtxt`, `np.savetxt`

et bien d'autres !

f2py

exemple.f95

```
SUBROUTINE addprod(a,b,c,d)
integer, intent(in) :: a, b
integer, intent(out) :: c, d
c = a + b
d = a * b
END SUBROUTINE addprod
```

Console

```
$ f2py -c -m mymod exemple.f95
```

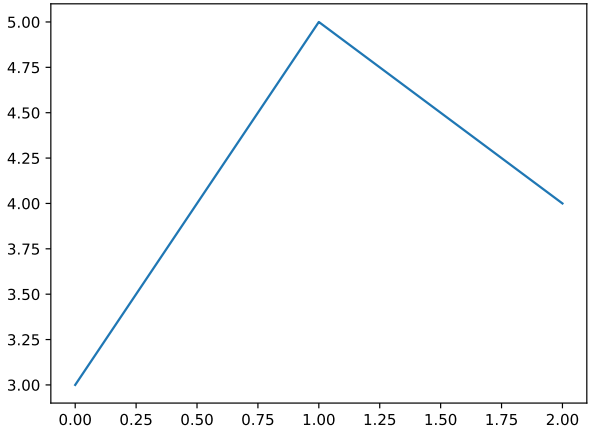
Python

```
>>> import mymod as m
>>> a, p = m.addprod(3,5)
>>> print(a,p)
 8  15
```

Permet de profiter

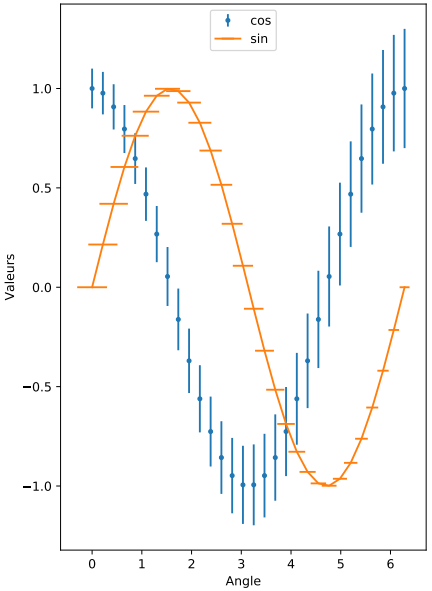
- de la vitesse de Fortran (compilé)
- de la facilité d'emploi de Python (interprété)

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([0,1,2],[3,5,4])
>>> plt.show()
```

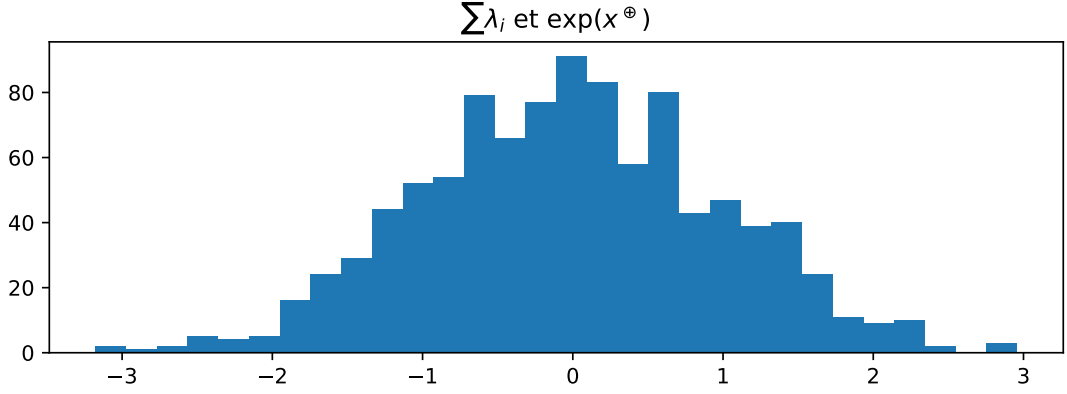


- Bibliothèque de référence pour les **graphiques**
- Syntaxe volontairement proche de **Matlab**
  - `plot`, `semilogx`, `semilogy`, `pcolormesh`, `contour`, `hist`, etc.
  - `figure`, `xlabel`, `ylabel`, `xticks`, `yticks`, `title`, `legend`, etc.
- Export au format JPG, PNG, PDF, etc.
- ≪ *Matplotlib tries to make easy things easy, and hard things possible.* ≫

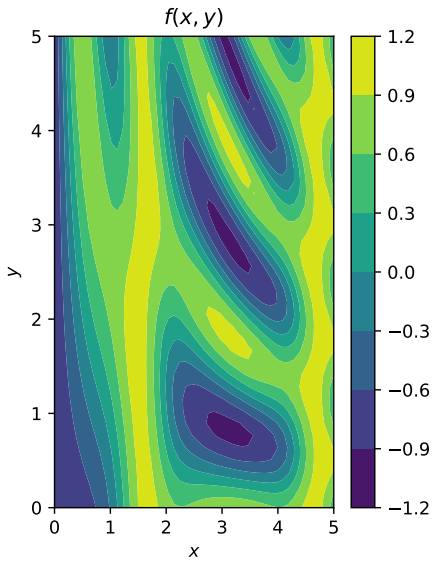
```
>>> angle= np.linspace(0.,2*np.pi,30)
>>> ye = np.linspace(0.1,0.3,30)
>>> plt.errorbar(angle,np.cos(angle),
    fmt='.',yerr=ye, label='cos')
>>> plt.errorbar(angle,np.sin(angle),
    xerr = ye[::-1], label='sin')
>>> plt.xlabel('Angle')
>>> plt.ylabel('Valeurs')
>>> plt.legend(loc='upper center')
>>> plt.show()
```



```
>>> dist = np.random.normal(0, size=1000)
>>> plt.hist(dist,30)
>>> plt.title('$\\sum \\lambda_i$ et $\\exp(x^\\oplus)$')
```



```
>>> def f(x,y):
...     a=np.sin(x)**10
...     b=np.cos(10+y*x)
...     return a+b*np.cos(x)
>>> x = np.linspace(0,5,50)
>>> y = np.linspace(0,5,40)
>>> X, Y = np.meshgrid(x,y)
>>> Z = f(X,Y)
>>> plt.contourf(X, Y, Z)
>>> plt.title("$f(x,y)$")
>>> plt.xlabel("$x$")
>>> plt.ylabel("$y$")
>>> plt.colorbar()
>>> plt.show()
```



scipy

- Traitement d'images `scipy.ndimage`
- Statistiques `scipy.stats`
- Lecture/écriture (IDL, Matlab, NetCDF, Fortran, etc.) `scipy.io`
- Interpolation avancée `scipy.interpolate`
- Intégration `scipy.integrate`
- Équations différentielles ordinaires `scipy.odeint`
- etc.

sympy Calcul symbolique

shapely Géométrie planaire

datetime Gestion des dates et heures (compatible avec Matplotlib)

astropy Astronomie et éphémérides

cartopy Projections cartographiques

arcpy, geopandas SIG

Liste loin d'être exhaustive : utilisez les **ressources en ligne**

- StackOverflow, galeries Matplotlib, tutoriels, etc.