CalculerAvecPython

Sébastien Joannès

September 18, 2015

Contents

1 Le calcul scientifique & Python

1.1 Qu'est-ce que Python ?

Python est un langage de programmation, créé en 1991 par Guido van Rossum (Pays-Bas) et a été baptisé ainsi en hommage à la troupe de comiques les **Monty Python**.
Python a rapidement voyagé du Macintosh de son créateur vers une version publique et en 2001, la **Python Software Fundation** est créée afin de développer et promouvoir ce langage de programmation.

- Un langage **objet**:

La POO pour **Programmation Orientée Objet**, consiste à définir et à assembler des briques logicielles appelées **objets** qui interagissent entre elles. En Python, ce paradigme (style de programmation) peut êre associé à d'autres méthodes d'élaboration (langage multi-paradigmes)

Figure 1: Logo Python

- Un langage **libre**:

Python est placé sous une licence proche de la licence BSD (Berkeley Software Distribution License) utilisée pour la distribution de logiciels. Elle permet de réutiliser tout ou une partie du logiciel sans restriction, qu'il soit intégré dans un logiciel libre ou propriétaire.

- Un langage **multi-plateformes**:

Python fonctionne sur la plupart des plates-formes informatiques, de Windows à Unix en passant par Mac OS, GNU/Linux, ou encore Android, iOS, ...

Python offre des outils de très haut niveau, c'est à dire qui permettent d'écrire des programmes en utilisant des mots usuels (haut degré d'abstraction). La syntaxe est simple à utiliser et à maîtriser. En Python, on construit un environnement de travail à l'aide de **modules** (**packages**) souvent basés sur des bibliothèques reconnues écrites en C++ ou Fortran.
Pour un environnement de calcul scientifique on fait généralement appel à 3 packages:

- `numpy`: opérations vectorisées pour la manipulation de matrices.
- `scipy`: bibliothèque scientifique complète, construite sur numpy.
- `matplotlib`: bibliothèque graphique.

1.2 Pourquoi Python ?

En cacul scientifique, Python est donc une alternative gratuite, libre et performante à Matlab, Octave, Scilab mais également à des outils de plus bas niveaux. Python peut coopérer avec un très grand nombre de composants logiciels, ce qui en fait un langage idéal pour mutualiser les ressources et les codes.

En résumé, l'engouement croissant pour Python réside essentiellement sur les points suivants:

- Simplicité du langage
- Facilité à interfacer
- Excellente portabilité
- Multi-disciplinarité et communauté très active
- ...

2 Premiers pas en Python

2.1 L'interpréteur de commande

Python peut être utilisé de plusieurs manières différentes. En mode **interactif**, il est possible de dialoguer avec lui directement depuis un **interpréteur de commande**. Cela permet de découvrir plus facilement un grand nombre de fonctionnalités offert par ce langage.
Un interpréteur de commande interprète et exécute les commandes qu'un utilisateur lui soumet. Il existe de nombreux interpréteurs du langage Python:

- CPython (interpréteur de base écrit en C)
- IPython (interpréteur interactif)
- Jython (écrit en Java)
- PyPy (écrit en Python)
- IDLE (environnement intégré, écrit avec Python & Tkinter)

La plupart des interpréteurs peuvent être utilisés en mode **interactif** (en ligne de commande). En Python, le mode interactif est signalé par le **prompt** >>>. On quitte l'interpréteur en passant la commande `quit()` ou `exit()`.

```
>>>
```

Le mode interactif donne accès à une une aide en ligne `help()` ou `help("module")`, `help("module.objet")`, etc.

```
>>> help()

Welcome to Python 2.7!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/2.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".
```

2.2 Quelques opérations de base

Les opérations de base renvoient directement le résultat après avoir appuyer sur la touche entrée.

```
>>> 3
```

```
3
```

```
>>> 3+4, 2*25

(7, 50)
```

La virgule est ici utilisée pour former une séquence mais peut également être employée pour des affectations simultanées.

La notation anglo-saxonne est requise pour les nombres décimaux et le point remplace alors la virgule:

```
>>> -5.5*8e-3

-0.044
```

Associé à des entiers, l'opérateur `/` renvoie la partie entière de la division tout comme `//`. Le modulo (reste) est obtenu par l'opérateur `%`:

```
>>> 1/3, 1//3, 1./3, 1.//3, 1%3, 1.%3

(0, 0, 0.3333333333333333, 0.0, 1, 1.0)
```

2.3 La déclaration des variables

En Python, les variables sont déclarées par `nom_variable = valeur`. Il existe une quinzaine de **types** différents que Python reconnait automatiquement lors de la création des variables. Attention certains mots clefs sont réservés comme `and`, `if`, `import`, `continue`, etc.
Les principaux types de variables Python sont détaillés dans le tableau suivant:

`bool`	Booléen
`int`	Nombre entier optimisé
`long`	Nombre entier
`float`	Nombre à virgule flottante
`complex`	Nombre complexe
`str`	Chaîne de caractère
`unicode`	Chaîne de caractère unicode
`tuple`	Liste de longueur fixe
`list`	Liste de longueur variable
`dict`	Dicionnaire
`file`	Fichier
`function`	Fonction
`module`	Module

```
>>> ma_variable = 3
... type(ma_variable)
```

```
int
```

```
>>> ma_variable = 3.5
... type(ma_variable)
```

```
float
```

Si les opérations intégrant des variables sont évidentes, l'élévation à une puissance donnée nécessite d'utiliser l'opérateur **:

```
>>> ma_variable**2+5.
```

```
17.25
```

Après les nombres entiers `int` et les réels `float`, les chaînes de caractères (`str`) et les listes (`list`) sont des exemples de types très régulièrement utilisés...

```
>>> ma_chaine = 'Ceci est une chaine de caracteres'
... type(ma_chaine)
```

```
str
```

```
>>> ma_liste = [1,2,3]
... type(ma_liste)
```

```
list
```

Une liste Python est bien plus qu'un tableau. Elle est indexée à partir de 0 et c'est un objet qui utilise des les méthodes de la classe `list` comme `append`, `extend`, `insert`, etc. Une liste est donc un objet capable de contenir d'autres objets de n'importe quel type.

```
>>> ma_liste = list() # On utilise le nom de la classe pour instancier un objet de cette classe
... ma_liste = []     # On peut également créer une liste vide de cette façon
... ma_liste = [1, 'a', 2.5]
... ma_liste.append('nouvelle entree')
... print ma_liste
...
... ma_liste.insert(2,-1)
... print ma_liste
...
... ma_liste.sort()
... print ma_liste
```

```
[1, 'a', 2.5, 'nouvelle entree']
[1, 'a', -1, 2.5, 'nouvelle entree']
[-1, 1, 2.5, 'a', 'nouvelle entree']
```

Il faut remarquer que dans l'exemple précédent, l'utilisation du caractère `#` permet d'insérer un commentaire qui n'est donc pas considérer par l'interpréteur comme une commande.

Parcourir une liste peut s'effectuer de nombreuses façons:

```
>>> print ma_liste[2:4]

[2.5, 'a']

>>> for elt in ma_liste:
...     print elt

-1
1
2.5
a
nouvelle entree
```

Ou encore en utilisant la fonction `enumerate`...

```
>>> for i, elt in enumerate(ma_liste):
...     print("À l'indice {} se trouve {}.".format(i, elt))

À l'indice 0 se trouve -1.
À l'indice 1 se trouve 1.
À l'indice 2 se trouve 2.5.
À l'indice 3 se trouve a.
À l'indice 4 se trouve nouvelle entree.
```

Il faut remarquer que dans les deux derniers exemples, une **indentation** a été utilisé pour définir le bloc `for`. En Python, chaque bloc de code (fonction, instruction if, boucle for ou while etc.) est défini par son indentation. L'indentation démarre le bloc et la désindendation le termine. Cela signifie que les espaces blancs sont significatifs et qu'ils doivent rester cohérents.

Les **tuples** sont également des séquences, semblables aux listes. Leur contenu est néanmoins figé lors de leur création et on ne peut plus ajouter ou retirer des éléments par la suite.

```
>>> mon_tuple = tuple() # Ou mon_tuple = ()
... mon_tuple = (3,6,1)
... print mon_tuple
...
... mon_tuple.append(4)

(3, 6, 1)
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-15-8f0e2b14da95> in <module>()
  3 print mon_tuple
  4
----> 5 mon_tuple.append(4)


AttributeError: 'tuple' object has no attribute 'append'
```

Si l'on souhaite créer un tuple contenant un unique élément, il est indispensable de mettre une virgule après celui-ci. Plusieurs commandes peuvent par ailleurs être écrites sur la même ligne en utilisant le point virgule ";".

```
>>> mon_tuple = (1); print type(mon_tuple)
... mon_tuple = (1,); print type(mon_tuple)
```

```
<type 'int'>
<type 'tuple'>
```

Un tuple peut par exemple servir à stocker les coordonnées d'un point ou les dimensions d'une matrice.

A l'instar des listes, les **dictionnaires** sont des objets pouvant en contenir d'autres. Cependant, au lieu d'héberger des informations dans un ordre précis, ils associent chaque objet contenu à une clé.

```
>>> # Un dictionnaire vide
... mon_dico = dict()
... mon_dico = {}
...
... # Un dictionnaire des couleurs RVB, utilisant également des tuples
... mon_dico["rouge"] = (255,0,0)
... mon_dico["vert"]  = (0,255,0)
... mon_dico["bleu"]  = (0,0,255)
... print mon_dico
```

```
{'bleu': (0, 0, 255), 'vert': (0, 255, 0), 'rouge': (255, 0, 0)}
```

```
>>> # Création en une ligne, la notion d'ordre est inutile
... mon_dico = {'rouge': (255, 0, 0), 'vert': (0, 255, 0), 'bleu': (0, 0, 255)}
... print mon_dico
```

```
{'bleu': (0, 0, 255), 'vert': (0, 255, 0), 'rouge': (255, 0, 0)}
```

```
>>> print mon_dico["rouge"]
... print mon_dico.keys() # Utilisation des méthodes de la classe 'dict' pour accéder aux données
... print mon_dico.values()
```

```
(255, 0, 0)
['bleu', 'vert', 'rouge']
[(0, 0, 255), (0, 255, 0), (255, 0, 0)]
```

2.4 Les structures conditionnelles

Une **structure conditionnelle**, permet de faire un choix sur le jeu d'instructions à exécuter suivant telle ou telle condition.

```
>>> a    = 1
... str_ = 'La valeur de "a" est '
... if (a > 0):
...     print str_ + 'positive'
... elif (a == 0):
...     print str_ + 'nulle'
... else:
...     print str_ + 'négative'
```

```
La valeur de "a" est positive
```

```
>>> if (a >= 0) and (a <= 1): # Application d'une double condition
...     if (a != .5):          # Ou d'une imbrication
...         print 'La valeur est bien celle attendue'
```

```
La valeur est bien celle attendue
```

En Python, il n'y a pas de `switch/case`, conditions qui peuvent effectivement être remplacées par une suite de `if` et `elif` ou par des dictionnaires.

2.5 Les boucles `for` et `while`

Un petit aperçu de l'utilisation des boucles `for` a été donné lors de la présentation des listes pour parcourir les éléments d'une séquence. Une boucle `while` permet quant à elle de répéter un bloc d'instructions tant qu'une condition est vraie.

```
>>> i = 1
... while i < 5:
...     i += 1
... print 'Terminé !'
```

```
Terminé !
```

```
>>> i = 1
... while 1:
...     if (i > 10):
...         break # L'instruction 'break' est très utile pour stopper les boucles infinies
...     i += 1
... print 'Terminé !'

Terminé !
```

2.6 Les fonctions et la modularité

Si l'interpéteur intéractif est très efficace, certaines instructions répétitives peuvent être écrites dans un script que l'interpréteur exécute ligne à ligne. Un **script** est donc un fichier texte qui peut êre écrit avec n'importe qu'elle éditeur. Cependant, un éditeur dédié facilite grandement cette phase d'implémentation en mettant par exemple en couleurs les mots clefs. Il existe de nombreuses solutions et environnements de développement Python qui intègrent cette fonctionnalité.

Les **fonctions** sont particulièrement bien adaptées pour découper le problème global en éléments plus simples. Une fonction Python se définit de la façon suivante...

```
def maFonction(paramètres):
    instructions
```

```
>>> def printBonjour(): # ... et peut être ramené à sa plus simple expression
...     print 'bonjour'
...
... print type(printBonjour)
...
... printBonjour() # Appel à la fonction

<type 'function'>
bonjour
```

Une fonction peut être un peu plus compliquée et renvoyer un résultat. Dans l'exemple suivant, l'ouverture d'un fichier est testée (existence) avant de retourner un objet de type `file` ou une erreur.

```
>>> def ficOpen_(ficName_,openRule_): # Ou être un peu plus compliquée
...     ## Open file command with messaging
...     try:
...         fic_ = open(ficName_,openRule_)
...         return fic_
...     except:
...         print 'ficOpen_() Error : '\
...             'IO problem with \'%s\''%(ficName_)
...         exit(1)
```

Si les fonctions sont nombreuses, elles peuvent être regroupées dans un **script** qui est enregistré dans un fichier d'extension `.py`. La fonction `printBonjour` peut par exemple être sauvegardé dans un fichier `mesFonctions.py`. L'appel à cette fonction nécessite de charger préalablement le fichier.

```
>>> import mesFonctions          # Chargement du fichier
... mesFonctions.printBonjour() # Appel de la fonction

bonjour
```

Si nous intégrons au fichier `mesFonctions.py` d'autres fonctions ou définitions (classes, etc.), nous obtenons un **module** qu'il est possible de charger dans un script ou de manière interactive.

Un **module** peut être chargé de différentes manières:

```
>>> import mesFonctions
... mesFonctions.printBonjour()

bonjour

>>> import mesFonctions as mf # Simplification
... mf.printBonjour()

bonjour

>>> from mesFonctions import * # Chargement dans l'espace de nom 'global'
... printBonjour()

bonjour

>>> from mesFonctions import printBonjour # Chargement d'une seule fonction
... printBonjour()

bonjour
```

3 Les modules `numpy`, `scipy` et `matplotlib`

3.1 Import et présentation des modules

Comme pour tous les modules Python, ces trois modules de calcul scientifique peuvent s'importer de la manière suivante:

```
>>> import numpy as np
... import scipy as sp
... import matplotlib as mpl
... import matplotlib.pyplot as plt
```

Le module `numpy` est une bibliothèque destinée à manipuler des matrices ou des tableaux multidimensionnels ainsi que des fonctions mathématiques opérant sur ces tableaux. **NumPy** est la base de **SciPy**, regroupement de bibliothèques Python autour du calcul scientifique. **Matplotlib** est un module destiné à tracer et visualiser des données sous formes de graphiques.

3.2 Tableaux de nombres

En mathématiques, une **matrice** à m lignes et n colonnes est un tableau rectangulaire (bidimensionnel) de $m \times n$ nombres, rangés ligne par ligne. Il existe donc des matrices à une **ligne** $1 \times n$ et des matrices à une **colonne** $m \times 1$. Les indices sont généralement notés $i \in \{1, \ldots, m\}$ et $j \in \{1, \ldots, n\}$. Les matrices permettent notamment de représenter les systèmes d'équations linéaires sous une forme particulièrement appropriée à leur résulution. En algèbre multilinéaire, la notion de **tenseur** permet de représenter des applications multilinéaires et une comparaison abusive considérerait un tenseur comme une généralisation à un nombre arbitraires d'indices $i, j, k, \ldots$ du concept de matrice carrée ($m = n$). La notion de **tenseur** est bien plus large, tout comme la notion de **vecteur** (tenseur d'ordre 1).
Néanmoins, un tenseur $T_{i,j,k,\ldots}$ peut effectivement être représenté par un tableau de nombres. Il convient surtout de correctement définir les opérations de l'algèbre tensoriel sur cet objet (une librairie comme PyTensor peut, le cas échéant, apporter les outils nécessaires). Ainsi, les composantes d'un vecteur pourront être rangées dans une matrice (colonne) de taille m tout comme les composantes d'un tenseur dans un tableau à $m \times n \times p \times \ldots$ dimensions.

Dans ce cours, les notations sont les suivantes:

a	$\bullet$	Scalaire, tenseur d'ordre 0		
A_i	$\underline{\bullet}$	Vecteur, tenseur d'ordre 1	Matrice colonne	$\{\bullet\}$
$A_{i,j}$	$\underline{\underline{\bullet}}$	Tenseur d'ordre 2	Matrice $m \times n$	$[\bullet]$
$A_{i,j,k,l}$	$\underline{\underline{\underline{\bullet}}}$	Tenseur d'ordre 4		

Une matrice ligne s'écrit naturellement $\{\bullet\}^T$, le T désignant ici la transposée. Une matrice est par définition **bidimensionnelle** contrairement à un tableau qui peut être **monodimensionnel**.

NumPy ajoute le type `ndarray` qui est similaire à une liste dont tous les éléments sont du même type (`int`, `float`, etc.) pour représenter les composantes d'un tenseur. Il est possible de déclarer un tel tableau en partant d'une liste:

```
>>> mon_array = np.array([1, 4, 5, 8], float) # Tableau à une seule dimension
... print type(mon_array)
... print mon_array

<type 'numpy.ndarray'>
[ 1.  4.  5.  8.]

>>> mon_array = np.array([[[1,2],[2,3]],[[4,5],[6,7]]])
... print mon_array.ndim  # Obtention de la dimension du tableau
... print mon_array.shape # Obtention de la taille du tableau

3
(2L, 2L, 2L)
```

Quelques précisions sur les dimensions...

```
>>> mon_array = np.array([1, 4, 5, 8]) # Tableau à une seule dimension
... print mon_array.ndim, mon_array.shape
...
... mon_array = np.array([[1, 4, 5, 8]]) # Matrice ligne
... print mon_array.ndim, mon_array.shape
...
... mon_array = np.array([[1], [4], [5], [8]]) # Matrice colonne
... print mon_array.ndim, mon_array.shape
... print mon_array
...
... print mon_array.transpose() # Transposée
```

```
1 (4L,)
2 (1L, 4L)
2 (4L, 1L)
[[1]
 [4]
 [5]
 [8]]
[[1 4 5 8]]
```

Le type `mat` permet de créer des matrices (tableaux monodimensionnels et bidimensionnels) à la Matlab. Il existe également une classe `matrix` qui hérite de `ndarray` et qui facilite les opérations matricielles.

```
>>> ma_matrice = np.mat('[1,2,3 ; 4,5,6]')
... print type(ma_matrice)
... print ma_matrice
```

```
<class 'numpy.matrixlib.defmatrix.matrix'>
[[1 2 3]
 [4 5 6]]
```

```
>>> np.matrix([[1,2,3],[4,5,6]])
```

```
matrix([[1, 2, 3],
        [4, 5, 6]])
```

Les `ndarray` NumPy peuvent être créées:

- à partir de listes comme dans les exemples précédents (indexation identique)
- en utilisant des fonctions dédiées, telles que `arange`, `linspace`, etc.
- en chargeant des fichiers de données (solution que nous étudierons ultérieurement)

```
>>> np.arange(1, 10, 3) # début, fin, pas
```

```
array([1, 4, 7])
```

```
>>> np.linspace(1, 10, 5) # début, fin, nombre de points

array([  1.  ,   3.25,   5.5 ,   7.75,  10.  ])
```

Certains mots clefs comme `eye`,`zeros`, `ones`, `diad`, ... permettent de définir des tableaux particuliers.

```
>>> np.eye(2) # ou np.eye(2,2) Matrice identité

array([[ 1.,  0.],
       [ 0.,  1.]])
```

```
>>> np.ones((2,3))

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

```
>>> np.diag([1,2,3])

array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

Quelques opérations sur les tableaux et les matrices.

```
>>> mon_array = np.array([[1, 4, 5, 8]], float) # Matrice ligne
... print mon_array + mon_array # Somme terme à terme
... print mon_array * mon_array # Produit terme à terme

[[  2.   8.  10.  16.]]
[[  1.  16.  25.  64.]]
```

```
>>> ma_matrice = np.matrix([[1, 4, 5, 8]], float)
... print ma_matrice.transpose().shape
... print ma_matrice.shape
... print ma_matrice.transpose() * ma_matrice # Produit matriciel
... print ma_matrice * ma_matrice.transpose() # Produit scalaire
```

```
(4L, 1L)
(1L, 4L)
[[  1.   4.   5.   8.]
 [  4.  16.  20.  32.]
 [  5.  20.  25.  40.]
 [  8.  32.  40.  64.]]
[[ 106.]]
```

Le produit matriciel peut être obtenu directement sur des `ndarray` ...

```
>>> mon_array.transpose() * mon_array

array([[  1.,   4.,   5.,   8.],
       [  4.,  16.,  20.,  32.],
       [  5.,  20.,  25.,  40.],
       [  8.,  32.,  40.,  64.]])
```

... mais pas le produit scalaire!

```
>>> mon_array * mon_array.transpose()

array([[  1.,   4.,   5.,   8.],
       [  4.,  16.,  20.,  32.],
       [  5.,  20.,  25.,  40.],
       [  8.,  32.,  40.,  64.]])


>>> np.dot(mon_array, mon_array.transpose())

array([[ 106.]])
```

Une documentation complète sur les fonctionnalités de NumPy est disponible dans le **NumPy Reference Guide** écrit par la communauté NumPy.

3.3 Une collection d'algorithmes et de fonctions

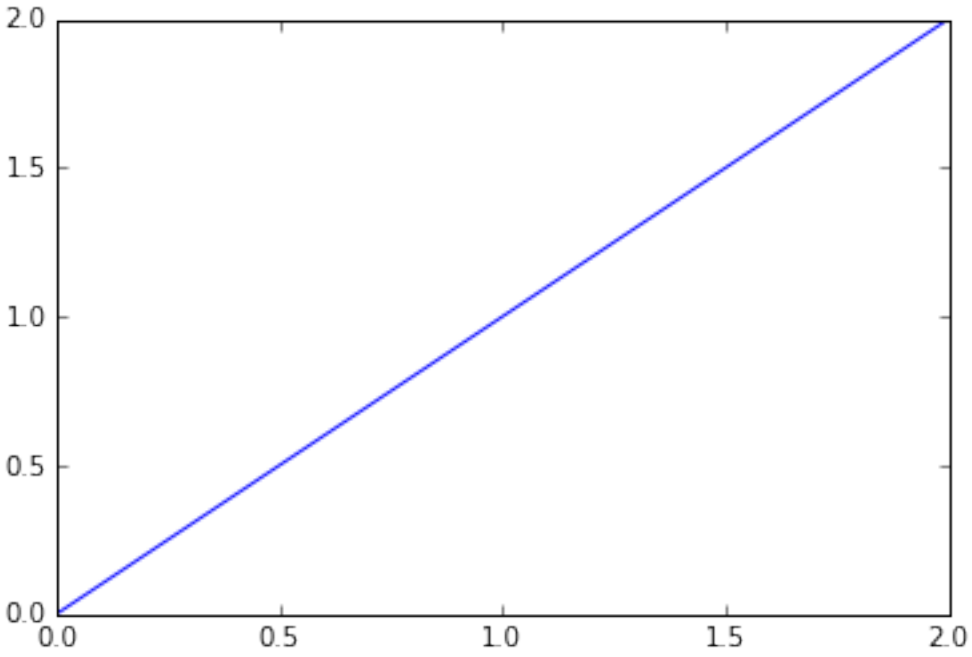
Tout comme NumPy, le module SciPy dispose de multiples fonctionnalités notamment décritent dans le **SciPy Reference Guide**.
De nombreux outils seront utilisés tout au long du cours comme par exemple:

- Interpolation (scipy.interpolate)
- Algèbre linéaire (scipy.linalg)
- Intégration (scipy.integrate)
- Optimisation (scipy.optimize)
- ...

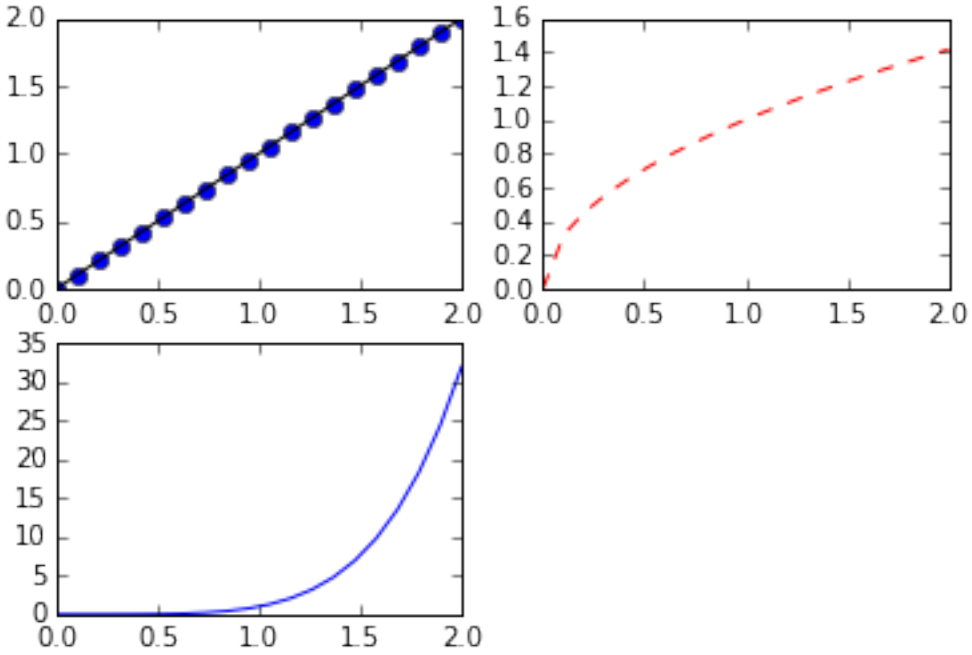
3.4 Créer des graphiques

`matplotlib.pyplot` est une collection de commandes qui offre un environnement très similaire à Matlab.
Des tableaux de points sont utilisés comme arguments.

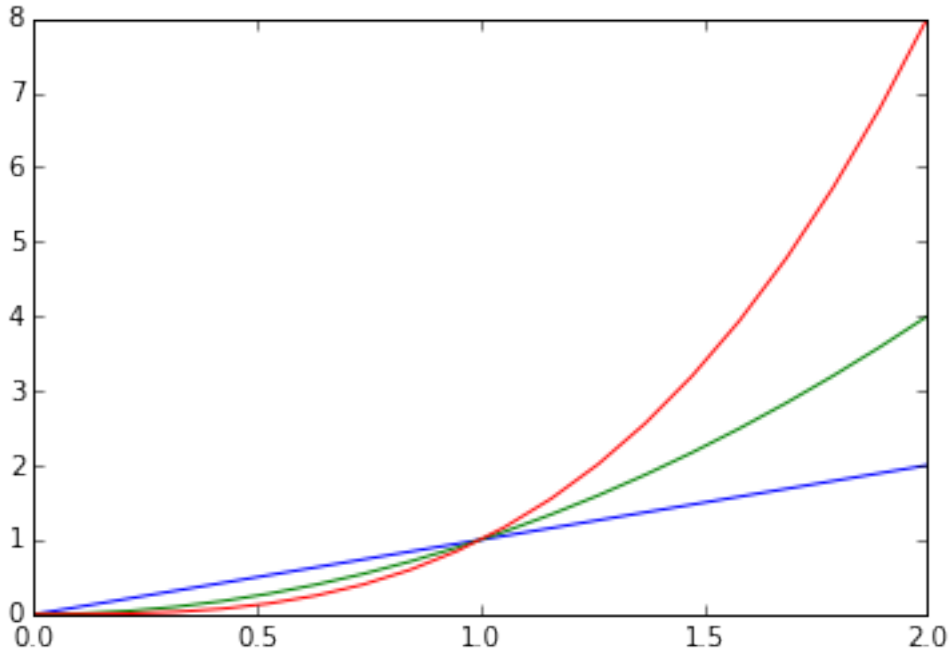
```
>>> x = np.linspace(0, 2, 20)
... plt.plot(x, x); plt.show()
```



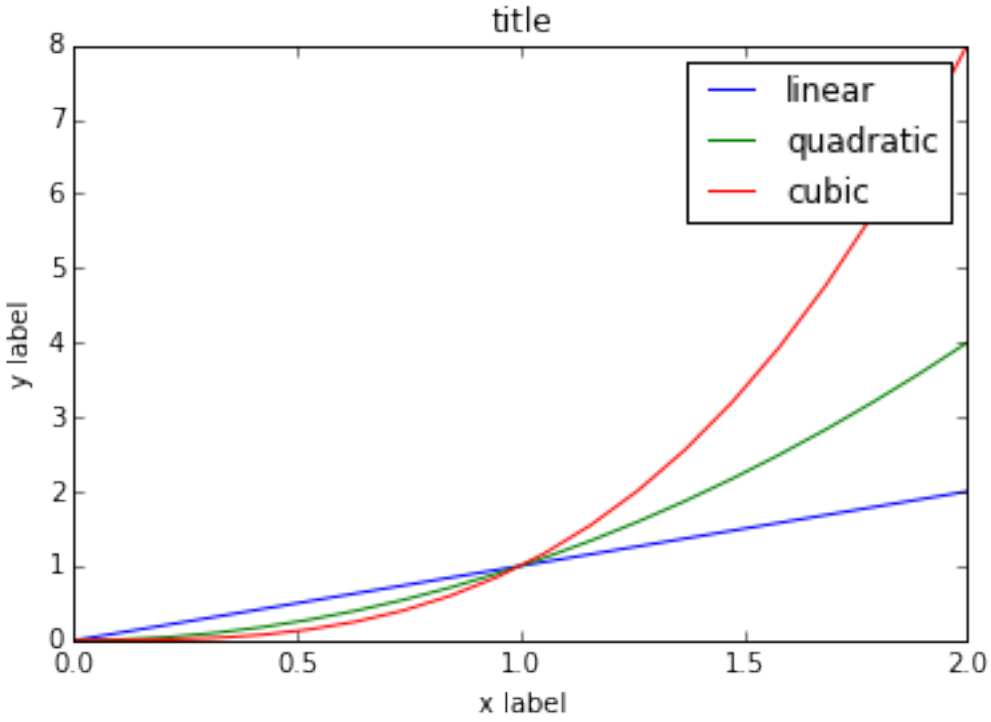

```
>>> plt.figure(1);      # Création d'une figure
... plt.subplot(221);  # Utilisation des 'subplots'
... plt.plot(x, x, 'bo', x, x, 'k');
... plt.subplot(222);
... plt.plot(x, x**.5, 'r--');
... plt.subplot(223);
... plt.plot(x, x**5, 'b-');
```



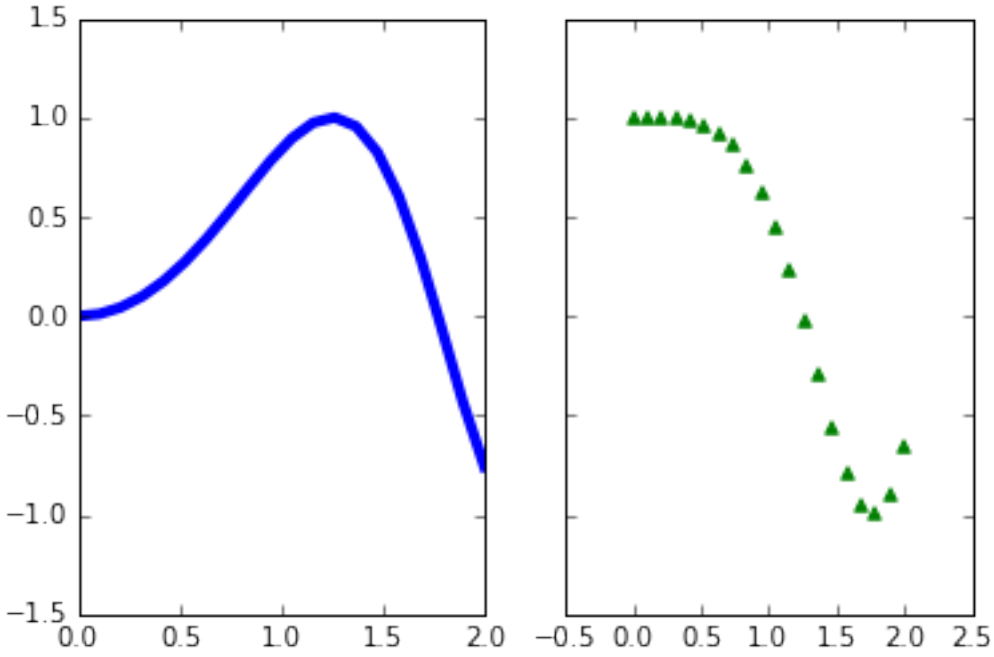

```
>>> fig, ax1= plt.subplots()                # Création d'une figure et ajout d'un axe sur la figure
... ax1.plot(x, x, label='linear');         # Placer les courbes sur l'axe
... ax1.plot(x, x**2, label='quadratic');
... ax1.plot(x, x**3, label='cubic');
```




```
>>> ax1.legend();
... ax1.set_xlabel('x label');
... ax1.set_ylabel('y label');
... ax1.set_title('title');
... fig
```

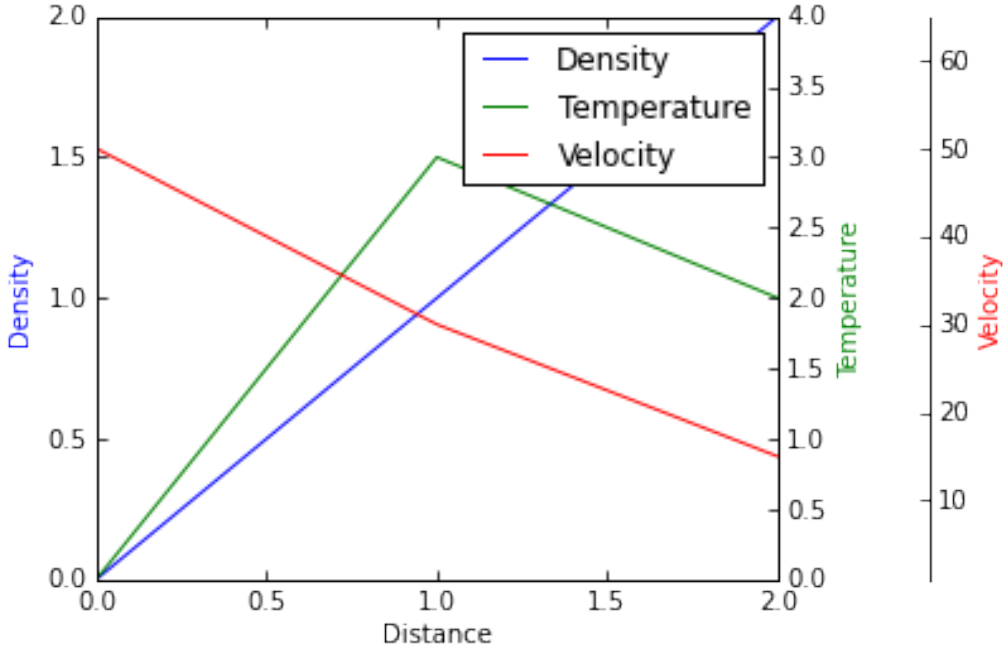



```
>>> fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True); # Partage du même axe y
... ax1.plot(x, np.sin(x**2), linewidth=4.0);
... ax2.scatter(x, np.cos(x**2), color='g', marker='^');
```

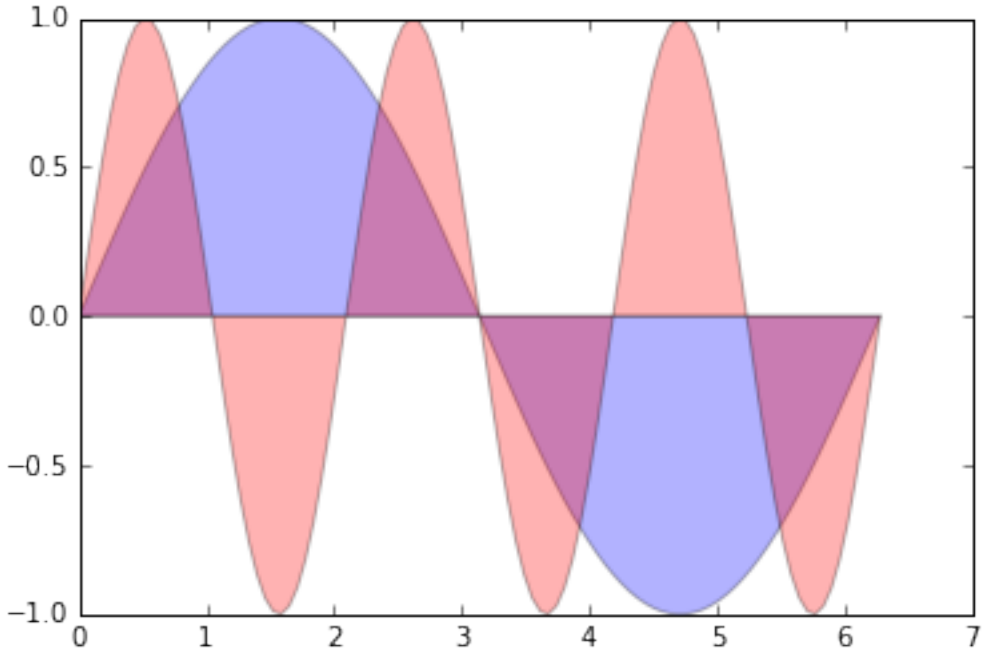


Quelques exemples tirés du site `http://matplotlib.org/examples`

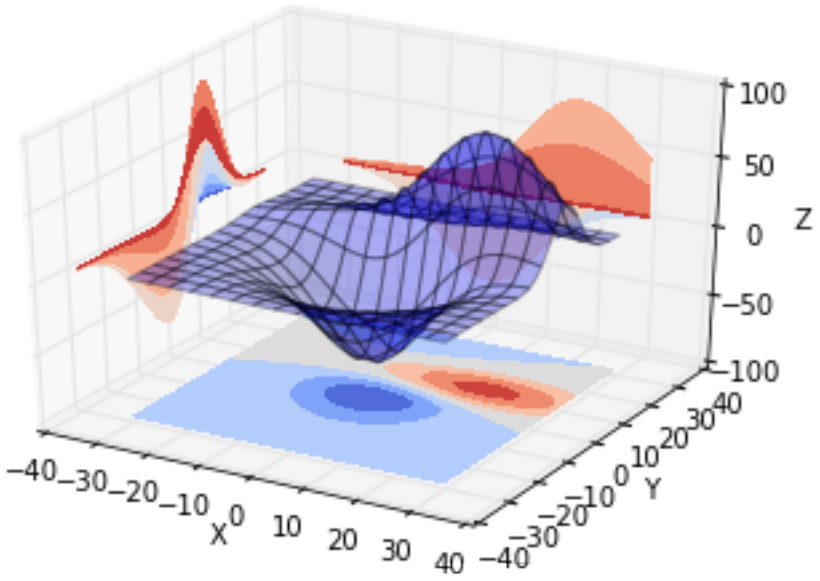
```
>>> import demo_parasite_axes2
```




```
>>> import fill_demo_features
```




```
>>> import contourf3d_demo2
```


100
50
0
−50
−100
Z
−40
−30
−20
−10
0
10
20
30
40
X
−40
−30
−20
−10
0
10
20
30
40
Y