Introduction à la programmation en langage Python

1. Présentation

Le Langage Python

Python est un langage de programmation (au même titre que le C, C++, fortran, java ...), développé en 1989. Ses principales caractéristiques sont les suivantes :
— «**open-source**» : son utilisation est gratuite et les fichiers sources sont disponibles et modifiables ;
— **simple et très lisible** ;
— doté d'une **bibliothèque de base très fournie** ;
— importante quantité de **bibliothèques disponibles** : pour le calcul scientifique, les statistiques, les bases de données, la visualisation ... ;
— **grande portabilité** : indépendant vis à vis du système d'exploitation (linux, windows, MacOS) ;
— **orienté objet** ;
— **typage dynamique** : le typage (association à une variable de son type et allocation zone mémoire en conséquence) est fait automatiquement lors de l'exécution du programme, ce qui permet une grande flexibilité et rapidité de programmation, mais qui se paye par une surconsommation de mémoire et une perte de performance ;
— présente un **support pour l'intégration d'autres langages**.

Comment faire fonctionner le code source ?

Il existe deux techniques principales pour traduire un code source en langage machine :
— la compilation : une application tierce, appelée compilateur, transforme les lignes de code en un fichier exécutable en langage machine. A chaque fois que l'on apporte une modification au programme, il faut recompiler avant de voir le résultat.
— l'interprétation : un interpréteur s'occupe de traduire ligne par ligne le programme en langage machine. Ce type de langage offre une plus grande commodité pour le développement, mais les exécutions sont souvent plus lentes.
Dans le cas de Python, on peut admettre pour commencer qu'il s'agit d'un langage interprété, qui fait appel des modules compilés. Pour les opérations algorithmiques coûteuses, le langage Python peut s'interfacer à des bibliothèques écrites en langage de bas niveau comme le langage C.

Les différentes versions

Il existe deux versions de Python : 2.7 et 3.3. La version 3.3 n'est pas une simple amélioration de la version 2.2. Attention, toutes les librairies Python n'ont pas effectué la migration de 2.7 à 3.3.

L'interpréteur

Dans un terminal, taper `python` (interpréteur classique) ou `ipython` (interpréteur plus évolué) pour accéder à un interpréteur Python. Vous pouvez maintenant taper dans ce terminal des instructions Python qui seront exécutées.

Utilisations de Python et librairies

— web : *Django, Zope, Plone,...*
— bases de données : *MySQL, Oracle,...*
— réseaux : *TwistedMatrix, PyRO, VTK,...*
— représentation graphique : *matplotlib, VTK,...*
— calcul scientifique : *numpy, scipy, ...*

Le mode programmation

Il s'agit d'écrire dans un fichier une succession d'instructions qui ne seront éffectuées que lorsque vous lancerez l'exécution du programme. Cela permet tout d'abord de sauvegarder les commandes qui pourront être utilisées ultérieurement, et d'autre part d'organiser un programme, sous-forme de fichier principal, modules, fonctions ....
Le fichier à exécuter devra avoir l'extension `.py`, et devra contenir en première ligne le chemin pour accéder au compilateur Python, ainsi que l'encodage :

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

Il pourra être exécuté en lançant dans un terminal la commande `python nomdufichier.py`.

L'environnement de programmation Spyder (sous Anaconda)

L'environnement Spyder permet de réaliser des programmes informatiques écrits avec le langage Python. Il est disponible avec la distribution Anaconda, qui présente de nombreux avantages, notamment celui d'être simple à installer. Son téléchargement se fait à l'adresse suivante : https ://store.continuum.io/cshop/anaconda/
Une fois la distribution Anaconda téléchargée et installée, on peut commencer à lancer Spyder en tapant `Spyder` dans un terminal.

Au lancement de Spyder, apparaît une fenêtre partagée en deux zones. La zone en-bas à droite, appelée console (shell en anglais), est celle où l'on peut travailler de façon interactive avec l'interprteur Python. La zone à gauche est un éditeur de texte, spécialement conçu pour écrire des programmes dans le langage Python.

2. Types et opérations de base

Le langage Python est orienté objet, c'est-à-dire qu'il permet de créer des objets, en définissant des attributs et des fonctions qui leur sont propres. Cependant, certains objets sont pré-définis dans le langage. Nous allons voir à présent les plus importants.

(a) **les nombres et les booléens**

— **entiers** (32 bits)
type : `int`
— **réels** (64 bits)
type : `float`
exemples de valeurs : `4.  5.1  1.23e-6`
— **complexes**
type : `complex`
exemples de valeurs : `3+4j  3+4J`
— **booléens** type : `bool`
exemples de valeurs : `True  False`

(b) **opérations de base**

— **affectation**

```
>>> i = 3    # i vaut 3
>>> a, k=True,  3.14159
>>> k=r=2.15
>>> x=complex(3,4)
```

— **affichage**

```
>>> i
3
>>> print(i)
3
```

— **Opérateurs addition, soustraction, multiplication, division**

`+, -,  *, \, \%,`

— **Opérateurs puissance, valeur absolue**

`**, pow, abs, \dots`

— **Opérateurs de comparaison**

`==, is, !=, is not, >, >=, <, <=`

— **Opérateurs logiques**

`or, and, not`

— **Conversion**

```
>>> int(3.1415)
3
>>> float(3)
3.
```

(c) **les chaînes de caractères**

Une chaîne de caractères (string en anglais) est un objet de la classe (ou de type) `str`. Une chaîne de caractères peut être définie de plusieurs façons :

```
 >>> "je suis une chaine"
'je suis une chaine'
>>> 'je suis une chaine'
'je suis une chaine'
```

```
>>> 'pour prendre l\'apostrophe'
'pour prendre l' apostrophe'

>>> "pour prendre l'apostrophe"
"pour prendre l'apostrophe"

>>> " " "ecrire
sur
plusieurs
lignes" " "
'ecrire\nsur\nplusieurs\nlignes'
```

— **concaténation**
On peut mettre plusieurs chaînes de caractères bout à bout avec l'opérateur binaire de concaténation, noté +.

```
>>> s = 'i vaut'
>>> i = 1
>>> print( s+i )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> print( s + " %d %s "%(i, "m."))
i vaut 1 m.
>>> print( s + ' ' + str(i))
i vaut 1
>>> print('*-' * 5)
*-*-*-*-*-
```

— **accès aux caractères**
Les caractères qui composent une chaîne sont numérotés à partir de zéro. On peut y accéder individuellement en faisant suivre le nom de la chaîne d'un entier encadré par une paire de crochets :

```
>>> "bonjour"[3]; "bonjour"[-1]
'j'
'r'
>>> "bonjour"[2:]; "bonjour"[:3]; "bonjour"[3:5]
'njour'
'bon'
'jo'
>>> "bonjour"[-1::-1];
'ruojnob'
```

— **méthodes propres**
  — **len(s)** : renvoie la taille d'une chaîne,
  — **s.find** : recherche une sous-chaîne dans la chaîne,
  — **s.rstrip** : enlève les espaces de fin,
  — **s.replace** : remplace une chaîne par une autre,
  — **...**

(d) **les listes**
Une liste consiste en une succession ordonnée d'objets, qui ne doivent pas nécessairement être du même type. Les termes d'une liste sont numérotés à partir de 0 (comme pour les chaînes de caractères).

— **initialisation**

```
>>> []; list();
[]
[]
>>> [1,2,3,4,5]; ['point','triangle','quad'];
[1, 2, 3, 4, 5]
['point', 'triangle', 'quad']
>>> [1,4,'mesh',4,'triangle',['point',6]];
[1, 4, 'mesh', 4, 'triangle', ['point', 6]]
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(2,10,2)
[2, 4, 6, 8]
```

— **modification**
Contrairement aux chaînes de caractères, on peut modifier les éléments d'une liste :

```
>>> l=[1,2,3,4,5]
>>> l[2:]=[2,2,2]
>>> l
[1, 2, 2, 2, 2]
```

— **concaténation**

```
>>> [0]*7
[0, 0, 0, 0, 0, 0, 0]
>>> L1, L2 = [1,2,3], [4,5]
>>> L1
[1, 2, 3]
>>> L2
[4, 5]
>>> L1+L2
[1, 2, 3, 4, 5]
```

— **méthodes propres**
- **len(L)** : renvoie la taille de la liste L,
- **L.sort** : trie la liste L,
- **L.append** : ajoute un élément à la fin de la liste L,
- **L.reverse** : inverse la liste L,
- **L.index** : recherche un élément dans la liste L,
- **L.remove** : retire un élément de la liste L,
- **L.pop** : retire le dernier élément de la liste L,
- **...**

(e) **copie d'un objet**

```
>>> L = ['Dans','python','tout','est','objet']
>>> T = L
>>> T[4] = 'bon'
>>> T
['Dans', 'python', 'tout', 'est', 'bon']
>>> L
['Dans', 'python', 'tout', 'est', 'bon']
>>> L=T[:]
>>> L[4]='objet'
>>> T;L
['Dans', 'python', 'tout', 'est', 'bon']
```

```
['Dans', 'python', 'tout', 'est', 'objet']
```

(f) **quelques remarques importantes**

— en Python, tout est objet,
— une chaîne de caractères est immuable, tandis qu'une liste est muable,
— **type** permet de connaître le type d'un objet,
— **id** permet de connaître l'adresse d'un objet,
— **eval** permet d'évaluer une chaîne de caractères.

3. **Commentaires**

```
# ceci est un commentaire
```

4. **Noms de variables**

Python fait la distinction entre minuscules et majuscules.
Conseil : donner des noms significatifs aux variables et aux fonctions.

5. **Les structures de contrôle**

(a) **L'indentation**

Les fonctions Python n'ont pas de `begin` ou `end` explicites, ni d'accolades qui pourraient marquer là où commence et où se termine le code de la fonction. Le seul délimiteur est les deux points (« : ») et l'indentation du code lui-même. Les blocs de code (fonctions, instructions if, boucles for ou while etc) sont définis par leur indentation. L'indentation démarre le bloc et la désindentation le termine. Il n'y a pas d'accolades, de crochets ou de mots clés spécifiques. Cela signifie que les espaces blancs sont significatifs et qu'ils doivent être cohérents. Voici un exemple :

```
a = -150
if a < 0:
    print('a est negatif')

ligne d'en-tête
    première instruction du bloc
    . . .
    dernière instruction du bloc
```

Fonctionnement par blocs :

```
Bloc 1
 ...
Ligne d'en-tête :
      Bloc 2
       ...
      Ligne d'en-tête :
            Bloc 3
             ...
             Ligne d'en-tête :
      Bloc 2 (suite)
       ...
Bloc 1 (suite)
 ...
```

(b) **Le test de conditions**

Le test de condition se fait sous la forme générale suivante :

```
if < test1 > :
      < blocs d'instructions 1>
elif < test2 > :
      < blocs d'instructions 2 >
else :
      < blocs d'instructions 3 >
```

Un exemple :

```
a = 10.
if a > 0:
      print('a est strictement positif')
      if a >= 10:
            print (' a est un nombre ')
      else:
            print (' a est un chiffre ')
            a += 1
elif a is not 0:
      print(' a est strictement negatif ')
else:
      print(' a est nul ')
```

Un autre exemple :

```
L = [1, 3, 6, 8]
if 9 in L:
      print '9 est dans la liste L'
else:
      L.append(9)
```

(c) **La boucle conditionnelle**

La forme générale d'une boucle conditionnelle est

```
while < test1 > :
      < blocs d'instructions 1 >
      if < test2 > : break
      if < test3 > : continue
else :
      < blocs d'instructions 2 >
```

où l'on a utilisé les méthodes suivantes :

break : sort de la boucle sans passer par else,
continue : remonte au début de la boucle,
pass : ne fait rien.
La structure finale **else** est optionnelle. Elle signifie que l'instruction est lancée si et seulement si la boucle se termine normalement.

Quelques exemples :

— Boucle infinie :

```
while 1:
      pass
```

— y est-il premier ?

```
x = y/2
while x > 1 :
      if y%x ==0
            print (str(y)+' est facteur de '+str(x))
            break
      x = x - 1
else :
     print( str(y)+ ' est premier')
```

(d) **La boucle inconditionnelle**

La forme générale d'une boucle inconditionnelle est

```
for < cible > in < objet > :
      < blocs d'instructions 1 >
      if < test1 > : break
      if < test2 > : continue
else :
      < blocs d'instructions 2 >
```

Quelques exemples :

```
sum = 0
for i in [1, 2, 3, 4] :
      sum += 1

prod = 1
for p in range(1, 10) :
      prod *= p

s = 'bonjour'
for c in s :
      print c,

L = [ x + 10 for x in range(10) ]
```

6. **Les fonctions**

La structure générale de définition d'une fonction est la suivante

```
def < nom fonction > (arg1, arg2,... argN):
          ...
          bloc d'instructions
          ...
          return < valeur( s ) >
```

Voici quelques exemples :

```
def table7():
          n=1
          while n < 11:
          print n*7
          n+=1
```

Un fonction qui n'a pas de `return` renvoie par défaut `None`.

```
def table(base):
          n=1
          while n < 11:
          print n*base
          n+=1


def table(base, debut=0, fin=11):
          print ('Fragment de la table de multiplication par '+str(base)+' : ')
          n=debut
          l=[]
          while n < fin:
          print n*base
          l.append(n*base)
          n+=1
          return l
```

On peut également déclarer une fonction sans connaître ses paramètres :

```
>>> def f (*args, **kwargs):
          ... print(args)
          ... print(kwargs)

>>> f(1,3,'b',j=1)
(1, 3, 'b')
{'j': 1}
```

Il existe une autre façon de déclarer une fonction de plusieurs paramètres :

```
lambda argument 1, ... , argument N : expression utilisant les arguments
```

Exemple :

```
>>> f = lambda x, i :  x**i
>>> f(2,4)
16
```

7. Les modules

Un module est un fichier comprenant un ensemble de définitions et d'instructions compréhensibles par Python. Il permet d'étendre les fonctionnalités du langage. Voici tout d'abord un exemple de module permettant l'utilisation des nombres de Fibonacci.

Fichier fibo.py

```
# Module nombres de Fibonacci
def print_fib(n) :
            " " "
            ecrit la serie de Fibonacci jusqu'a n
            " " "
            a, b = 0, 1
            while b < n:
                  print(b),
```

```
                a, b = b, a + b
        print

def print_fib(n) :
        " " "
        retourne la serie de Fibonacci jusqu'a n
        " " "
        result, a, b = [], 0, 1
        while b < n:
                result.append(b),
                a, b = b, a + b
        return result
```

Utilisation du module fibo.py

```
>>> import fibo
>>> fibo.print_fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.list_fib(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

L'importation de modules

Il existe plusieurs façons d'importer un module :

— **import** fibo
— **import** fibo **as** f
— **from** fibo **import** print_ fib, list_ fib
— **from** fibo **import** * (importe tous les noms sauf les variables et les fonctions privées)

Le module math

Ce module fournit un ensemble de fonctions mathématiques pour les réels :

```
pi
sqrt
cos, sin, tan, acos, ...
...
```

Tapez `import math` pour importer le module, puis `help(math)` pour avoir toutes les informations sur le module : chemin, fonctions, constantes.

8. **Le module NumPy**

NumPy est un outil performant pour la manipulation de tableaux à plusieurs dimensions. Il ajoute en effet le type `array`, qui est similaire à une liste, mais dont tous les éléments sont du même type : des entiers, des flottants ou des booléens.

Le module NumPy possède des fonctions basiques en algèbre linéaire, ainsi que pour les transformées de Fourier.

(a) **Création d'un tableau dont on connaît la taille**

```
>>> import numpy as np
>>> a = np.zeros(4)
>>> a
```

```
array([ 0., 0., 0., 0.])
>>> nx, ny = 2, 2
>>> a=np.zeros((nx,ny))
>>> a
array( [ [ 0.,  0. ] ,
         [ 0.,  0.] ] )
```

Un tableau peut être multidimensionnel, comme ici, de dimension 3 :

```
>>> a=np.zeros((nx,ny,3))
>>> a
array( [ [ [ 0.,  0.,  0. ],
           [ 0.,  0.,  0. ] ],

         [ [ 0.,  0.,  0. ],
           [ 0.,  0.,  0. ] ] ] )
```

Mais nous nous limiterons dans ce cours aux tableaux uni et bi-dimensionnels.

Il existe également les fonctions `np.ones`, `np.eye`, `np.identity`, `np.empty`, ...
Par exemple `np.identity(3)` est la matrice identité d'ordre 3, `np.empty` est un tableau vide, `np.ones(5)` est le vecteur `[1 1 1 1 1]` et `np.eye(3,2)` est la matrice à 3 lignes et 2 colonnes contenant des 1 sur la diagonale et des zéro partout ailleurs.

Par défaut les éléments d'un tableau sont des `float` (un réel en double précision) ; mais on peut donner un deuxième argument qui précise le type (`int, complex`, `bool`, ...). Exemple :

```
>>> np.eye(2, dtype=int)
```

(b) **Création d'un tableau avec une séquence de nombre**

La fonction `linspace(premier, dernier, n)` renvoie un tableau unidimensionnel commençant par `premier`, se terminant par `dernier` avec `n` éléments régulièrement espacés. Une variante est la fonction `arange` :

```
>>> a = np.linspace(-4, 4, 9)
>>> a
array([-4., -3., -2., -1., 0., 1., 2., 3., 4.])
>>> a = np.arange(-4, 4, 1)
>>> a
array([-4, -3, -2, -1, 0, 1, 2, 3])
```

On peut convertir une ou plusieurs listes en un tableau via la commande `array` :

```
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> b = np.array(range(10))
>>> b
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> L1, L2 = [1, 2, 3], [4, 5, 6]
>>> a = np.array([L1, L2])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
```

(c) **Création d'un tableau à partir d'une fonction**

```
>>> def f(x, y):
... return x**2 + np.sin(y)
...
>>> a = np.fromfunction(f, (2, 3))
>>> a
array([[ 0. , 0.84147098, 0.90929743],
[ 1. , 1.84147098, 1.90929743]])
```

La fonction `f` a ici été appliquée aux valeurs $x = [0, 1]$ et $y = [0, 1, 2]$, le résultat est

$$[\ [f(0,0), f(0,1), f(0,2)], [f(1,0), f(1,1), f(0,2)]\ ].$$

(d) **Manipulations d'un tableau**

i. Caractéristiques d'un tableau
   - **a.shape** : retourne les dimensions du tableau
   - **a.dtype** : retourne le type des éléments du tableau
   - **a.size** : retourne le nombre total d'éléments du tableau
   - **a.ndim** : retourne la dimension du tableau (1 pour un vecteur, 2 pour une matrice)

ii. Indexation d'un tableau

Comme pour les listes et les chaines de caractères, l'indexation d'un vecteur (ou tableau de dimension 1) commence à 0. Pour les matrices (ou tableaux de dimension 2), le premier index se réfère à la ligne, le deuxième à la colonne. Quelques exemples :

```
>>> L1, L2 = [1, -2, 3], [-4, 5, 6]
>>> a = np.array([L1, L2])
>>> a[1, 2]  #  extrait l'élément en 2ème ligne et 3ème colonne
6

>>> a[:, 1]  # extrait toute la deuxième colonne
array([-2, 5])

>>> a[1,0:2:2] # extrait les éléments d'indice [dbut=0, pas=2, fin=2]
               # de la première ligne, le dernier élément n'est pas inclus
array([-4])

>>> a[:, -1:0:-1]  # extrait les éléments d'indice [dbut=-1, pas=-1, fin=0]
                   # de la première colonne, le dernier élément n'est pas inclus
array([[3, -2],
[6, 5]])

>>> a[a < 0]    # extrait les éléments négatifs
array([-2, -4])
```

iii. Copie d'un tableau

Un tableau est un objet. Si l'on affecte un tableau A à un autre tableau B, A et B font référence au même objet. En modifiant l'un on modifie donc automatiquement l'autre. Pour éviter cela on peut faire une copie du tableau A dans le tableau B moyennant la fonction `copy()`. Ainsi, A et B seront deux tableaux identiques, mais ils ne feront pas référence au même objet.

```
>>> a = np.linspace(1, 5, 5)
>>> b = a
>>> c = a.copy()
>>> b[1] = 9
>>> a; b; c;
array([ 1., 9., 3., 4., 5.])
array([ 1., 9., 3., 4., 5.])
array([ 1., 2., 3., 4., 5.])
```

Une façon de faire une copie de tableau sans utiliser la méthode `copy()` est la suivante :

```
>>> d = np.zeros(b.shape, a.dtype)
>>> d[:] = b
>>> d
array([ 1., 2., 3., 4., 5.])
```

iv. Redimensionnement d'un tableau

Voici comment modifier les dimensions d'un tableau :

```
>>> a = np.linspace(1, 10, 10)
>>> a
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
>>> a.shape = (2, 5)
>>> a
array([[ 1., 2., 3., 4., 5.],
[ 6., 7., 8., 9., 10.]])
>>> a.shape = (a.size,)
>>> a
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
>>> a.reshape(2, 5)
array([[ 1., 2., 3., 4., 5.],
[ 6., 7., 8., 9., 10.]])
>>> a
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
```

Attention : la fonction `reshape` ne modifie pas l'objet, elle crée une nouvelle vue.
La fonction `flatten` renvoie une vue d'un tableau bi-dimensionnel en tableau uni-dimensionnel.

```
>>> a = np.linspace(1, 10, 10)
>>> a.shape = (2, 5)
>>> a
array([[ 1., 2., 3., 4., 5.],
[ 6., 7., 8., 9., 10.]])
>>> a.flatten
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
```

v. Boucles sur les tableaux

```
>>> a = np.zeros((2, 3))
>>> for i in range(a.shape[0]):
...     for j in range(a.shape[1]):
...         a[i, j] = (i + 1)*(j + 1)
>>> print a
[[ 1. 2. 3.]
[ 2. 4. 6.]]
```

```
>>> for e in a:
... print e
[ 1. 2. 3.]
[ 2. 4. 6.]
```

(e) **Calculs avec des tableaux**

Pour bien des calculs il est possible d'éviter d'utiliser des boucles (qui sont très consommatrices de temps de calcul). On peut faire directement les calculs sur des tableaux. Ceux-ci sont faits via des fonctions C, un langage de programmation bas niveau, et sont donc plus rapides. C'est ce qu'on appelle «vectoriser» un programme.

Voici un exemple, dans lequel on veut calculer $b = 3a - 1$:

```
>>> a = np.linspace(0, 1, 1E+06)
>>> %timeit b = 3*a -1
100 loops, best of 3: 10.7 ms per loop
>>> b = np.zeros(1E+06)
>>> %timeit for i in xrange(a.size): b[i] = 3*a[i] - 1
10 loops, best of 3: 1.31 s per loop
```

Voici un autre exemple de calcul fait directement sur un tableau :

```
>>> def f1(x):
...   return np.exp(-x*x)*np.log(1+x*np.sin(x))
...
>>> x = np.linspace(0, 1, 1e6)
>>> a = f1(x)
```

(f) **Produit matriciel**

Le calcul matriciel se fait avec la fonction `dot()`.
Attention, si le deuxième argument est un vecteur ligne, il sera transformé si besoin en vecteur colonne, par transposition. Ceci est dû au fait qu'il est plus simple de définir un vecteur ligne, par exemple `v=np.array([0,1,2])`, qu'un vecteur colonne, ici `v=np.array([[0],[1],[2]])`.
Exemple :

```
>>> A=np.array([[1,1,1],[0,1,1],[0,0,1]]
>>> v=np.array([0,1,2])
>>> np.dot(v,A)
array([0, 1, 3])
>>> np.dot(A,v)
array([3, 3, 2])
```

On peut aussi transposer préalablement un vecteur ligne :

```
>>> v=np.array([[0,1,2]])
>>> v=np.transpose(v)
>>> v
```

Attention, l'opération `A**2` correspond à une élévation au carré terme à terme. Pour lever une matrice au carré il faut taper `np.dot(A,A)`.

Pour lever une matrice carrée à une puissance n il faut faire une boucle :

```
>>> B=A.copy()
>>> for i in range(1,n):
...   B = np.dot(A,B)
```

A la sortie de cette boucle B contiendra A^n.

(g) **Produit scalaire**

Il faut utiliser la fonction `vdot()`.

(h) **Le sous-module linalg**

Ce module propose des méthodes numériques pour inverser une matrice, calculer son déterminant, résoudre un système linéaire ...

9. **Importer et exporter des données avec Python**

(a) Méthode manuelle

Pour ouvrir et fermer un fichier nommé `nomfichier.txt` on crée une variable `fichier` de type `file` à travers laquelle on pourra accéder au fichier.

```
fichier = open ('nomfichier.txt','r')
```

Le paramètre «r» indique qu'on accède au fichier en mode lecture. Pour accéder au fichier en mode écriture il faut utiliser le paramètre «w». A la fin des opérations sur le fichier, on le ferme en appelant la fonction `close` :

```
fichier.close()
```

Voici trois façons différentes de parcourir les lignes d'un fichier ouvert en mode écriture :

```
for ligne in fichier :
```

La variable ligne est une chaîne de caractère qui prendra les valeurs successives des lignes du fichier de lecture (sous la forme d'une chaîne de caractères).

```
ligne = fichier.readline()
```

A chaque fois que la fonction `readline` de fichier est appelée, la ligne suivante du fichier est lue et constitue la valeur de retour de la fonction.

```
lignes = fichier.readlines ()
```

Cette fonction lit toutes les lignes d'un coup et la valeur de retour est une liste de chaînes de caractères. Attention, si le fichier est gros cette méthode est à éviter car elle bloque instantannément une grande place mémoire. Il est dans ce cas préférable d'utiliser une méthode de lecture caractère par caractère (grâce à la fonction `read()`) ou ligne par ligne.

Enfin, la fonction `fichier.write(chaine)` permet d'écrire la chaîne de caractères en argument dans un fichier ouvert en mode écriture.

(b) Méthode automatisée

La fonction `numpy.loadtxt` lit les lignes du fichier en argument et renvoie une matrice dont les lignes correspondent aux lignes du fichier et les colonnes aux différentes valeurs délimitées par un espace.

Parmi les options les plus utiles, `skiprows` est le nombre de lignes à ne pas lire dans l'en–tête (par défaut 0) et `delimiter` remplace le délimiteur (espace) par la chaîne que l'on souhaite.

Exemple : le fichier `data.dat` contient les lignes :

```
chat, chien
3,4
2,5
1,8
```

On veut supprimer la première ligne qui contient le nom des colonnes mais pas de valeurs numériques, et on déclare que le délimiteur des valeurs est la virgule :

```
t = np.loadtxt ("data.dat", skiprows=1, delimiter=',')
t
```

renvoie

```
array([ [ 3., 4.],
    [ 2., 5.],
    [ 1., 8.]])
```

10. Représentation discrète d'une fonction avec Matplotlib

Pour tracer la courbe représentative d'une fonction f avec Matplotlib, il faut tout d'abord la "discrétiser" ; c'est-à-dire définir une liste de points $(x_i, f(x_i))$ qui va permettre d'approcher le graphe de la fonction par une ligne brisée. Bien sûr, plus on augmente le nombre de points, plus la ligne brisée est "proche" du graphe (en un certain sens).

Plus précisément, pour représenter le graphe d'une fonction réelle f définie sur un intervalle $[a, b]$, on commence par construire un vecteur X, discrétisant l'intervalle $[a, b]$, en considérant un ensemble de points équidistants dans $[a, b]$. Pour ce faire, on se donne un naturel N *grand* et on considère `X=numpy.linspace(a,b,N)` (ou, ce qui revient au même, on pose $h = \frac{b-a}{N-1}$ et on considère
`X=numpy.arange(a,b+h,h)`).

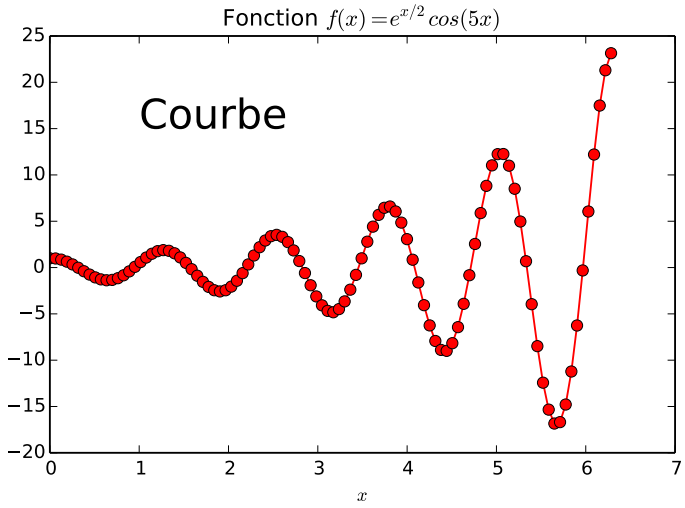
On construit ensuite le vecteur Y dont les composantes sont les images des X_i par f et on trace la ligne brise reliant tous les points (X_i, Y_i).

(a) La fonction `plot` du module matplotlib.pyplot

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

x = np.linspace(0., 2*np.pi, 100)
plt.plot(x, np.exp(x/2)*np.cos(5*x), '-ro')

plt.title('Fonction $f(x)=e^{x/2} cos(5x)$')
plt.xlabel('$x$')
plt.text(1, 15, 'Courbe', fontsize=22)
plt.savefig('1D_exemple.pdf')
plt.show()
```

Cette fonction permet de tracer des lignes brisées. Si X et Y sont deux vecteurs-lignes (ou vecteurs-colonnes) de même taille n, alors `plt.plot(X,Y)` permet de tracer la ligne brisée qui relie les points de coordonnées (X(i),Y(i)) pour $i = 1, \ldots, n$.

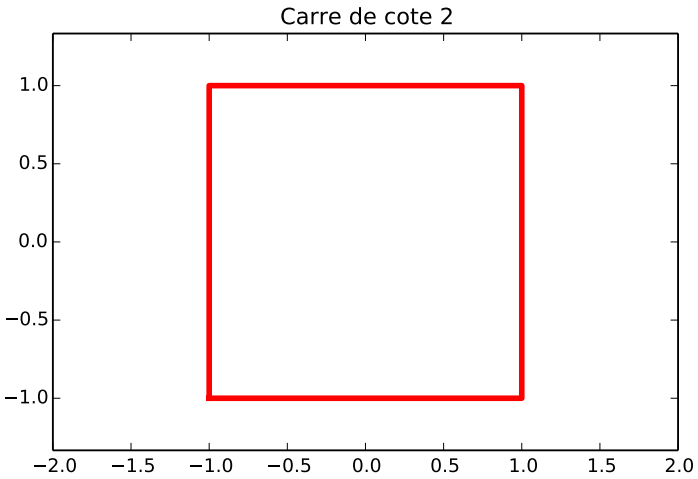
Pour connaître toutes les options, le mieux est de se référer à la documentation de Matplotlib.



Exemple : tracé d'un carré

Coordonnées des sommets : $X = (-1; 1; 1; -1; -1)$ et $Y = (-1; -1; 1; 1; -1)$.

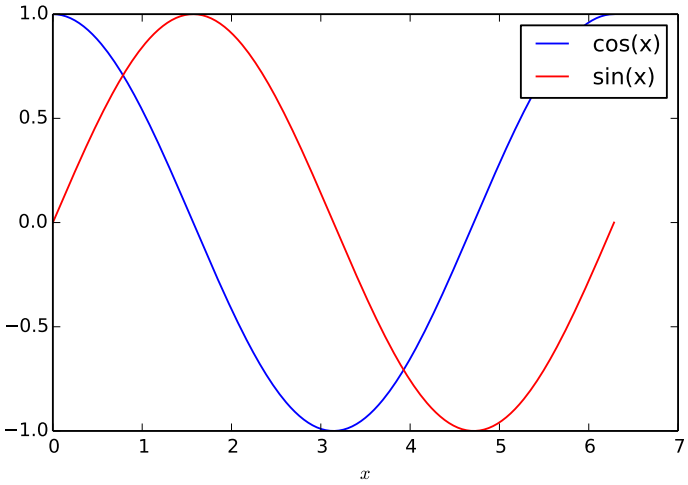
```
X=np.array([-1, 1, 1,-1,-1])
Y=np.array([-1, -1, 1,1,-1])
plt.plot(X,Y,'r',lw=3)
plt.axis('equal')
plt.axis([-2,2,-2,2])

plt.title('Carre de cote 2')
plt.savefig('carre_exemple.pdf')
plt.show()
```



(b) **Tracé de plusieurs lignes brisées**

On peut superposer plusieurs courbes dans le même graphique.

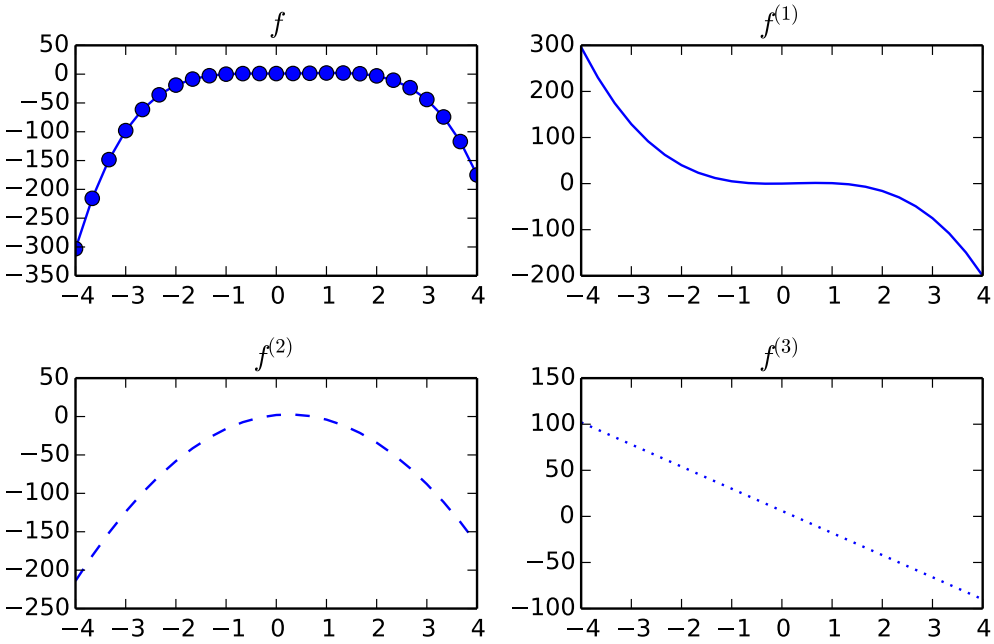
```
import matplotlib.pyplot as plt
import numpy as np
X = np.linspace(0, 2*np.pi, 256)
Ycos = np.cos(X)
Ysin = np.sin(X)
plt.plot(X,Ycos,'b')
plt.plot(X,Ysin,'r')
pp.show()
```



(c) **Création de plusieurs graphiques dans une même fenêtre**

La commande `subplot` permet de découper la fenêtre graphique en plusieurs sous-fenêtres. Voici un exemple d'utilisation de cette commande.

```
x = np.linspace(-4., 4., 25)
plt.subplot(2, 2, 1)
plt.plot(x, -x**4 + x**3 + x**2 + 1, 'o-')
plt.title("$f$")
plt.subplot(2, 2, 2)
plt.plot(x, -4*x**3 + 3*x**2 + 2*x, '-')
plt.title("$f^{(1)}$")
plt.subplot(2, 2, 3)
plt.plot(x, -12*x**2 + 6*x + 2, '--')
plt.title("$f^{(2)}$")
plt.subplot(2, 2, 4)
plt.plot(x, -24*x + 6, ':')
plt.title("$f^{(3)}$")
plt.tight_layout()
plt.savefig("1D_subplot2.pdf")
plt.show()
```



Matplotlib dispose d'autres commandes de tracé comme `loglog, polar`...