Table des matières

Soit on lance l'interpréteur via `python`, soit on exécute un script python, qui doit commencer par les lignes suivantes

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

La première ligne pour définir que c'est un script python et la deuxième pour spécifier l'encodage du script (pour moi, utf-8).

1 Pour débuter

En python, la mise en forme du code est primordiale puisque c'est elle qui détermine comment il est exécuté. Il est conseillé d'utiliser une indentation de 4 espaces (de n'utiliser que des espaces ou que des tabulations, mais une préférences pour les espaces, ne me demandez pas pourquoi...)

1.1 Boucles

Pour effectuer une boucle, on fera donc

```
for i in range(5):
  print i
print "ici, on n'est plus dans la boucle"
```

```
list = ['a', 'b', 'c', 'd']
for ele in list:
  print ele
```

```
a = 0
while (a<1):
  print a
  a += 1
```

```
a = 0
if (a==0):
  print "a est nul"
elif a==1:
  print "a vaut 1"
else:
  print "a ne vaut ni 0 ni 1"
# fin de la boucle
```

Opérateur : effet
a == b : teste si a et b sont égaux
a > b : teste si a est strictement supérieur à b
a < b : teste si a est strictement inférieur à b
a >= b : teste si a est supérieur ou égal à b
a <= b : teste si a est inférieur ou égal à b
a != b : teste si a et b sont différent

TABLE 1 – Quelques opérateurs de comparaison en python. Ils renvoient des expressions booléennes. On peut aussi combiner plusieurs tests à l'aide de `and` et `or`.

Remarque : À noter les fonction `continue` et `break` à utiliser dans des boucles. `continue` permet de sauter la boucle actuelle pour passer à l'itération suivante. `break`, quant à lui, permet d'arrêter la boucle avant sa fin.

1.2 Variables et types numériques

La fonction *type* permet de connaître le type d'une variable.

```
a = 2
b = 2.
c = 'chaine'
type(a), type(b), type(c)
```

Pour limiter le nombre de chiffres significatifs d'un réel, il faut utiliser la fonction `round(x,n)` qui arrondit la variable x avec n chiffres après la virgule.

```
>>> round(1.23441, 2)
1.23
```

Une fonction très pratique est la fonction `eval`. Elle permet de dire à python de considérer le contenu d'une variable ou expression comme si c'était du code python. Un exemple concrêt :

```
a = 2
b = 'a'
print eval(b)
```

On voit rapidement les possibilités de cette commande dans des boucles et autres joyeusetées.

Il existe en python le type booléen dont les valeurs possibles sont `True` ou `False`

```
a = True
if a:
  print a
a = False
if not(a):
  print "C'est faux"
```

1.3 Débug

Pour celà, il existe le module *pdb* qui sert à ça.
On l'importe de la façon suivante

```
import pdb
```

La commande utile, si on ne doit en retenir qu'une est :

```
pdb.set_trace()
```

Cette commande stoppe le programme et donne la main à l'interpréteur, nous permettant de regarder l'état des variables. Il permet aussi de continuer le programme ligne à ligne, mais je sais pas encore comment.

Ensuite, on peut taper des commandes python, notamment pour évaluer les variables. Mais il existe aussi des commandes propres au débuguage donc voici une liste des plus utiles :

- s continue l'exécution et s'arrête dès que possible (en gros, si tout va bien, s'arrête à la ligne suivante)
- r continue l'exécution jusqu'à la fin de la fonction en cours (s'arrête juste avant le return s'il y en a un il me semble)

Sinon, pour quitter, Ctrl+C, ou `exit()`

2 Fonctions

On peut bien évidemment créer des fonctions en python. Une fonction très simple serait

```
def affiche(variable):
  print variable
```

qui se contenterait d'afficher le contenu de la variable passée en argument.

Une fonction peut retourner des valeurs via `return variable`. Cette fonction peut renvoyer plusieurs valeurs, on aura donc :

```
def produit_somme(a,b):
  somme = a + b
  produit = a * b
  return somme, produit_somme

(sum_ab, prod_ab) = produit_somme(2,3)
```

On peut aussi avoir des paramètres optionnels, à qui on donne une valeur par défaut :

```
def test(a, b=3)
  print a*b
```

On peut aussi gérer un nombre inconnu de variable :

```
def test(*args):
  for arg in args:
    print arg
test('coucou', 2, 1, range(2))
test('coucou', 2)
```

Ceci peut s'ajouter à d'autres variables, elles obligatoires :

```
def test(nom, *variables):
  print nom
  for var in variables:
    print var
test('coucou', 2, 1, range(2))
test('coucou', 2)
```

2.1 Documenter une fonction

source : tuto de suzy

Une dernière chose très importante lorsque vous définissez une fonction, c'est sa documentation ! Vous allez commencer à documenter votre code pour le rendre plus compréhensible et exploitable par d'autres personnes que vous. Pour cela, c'est très simple, il suffit de placer immédiatement sous la ligne de définition de votre fonction, une chaîne de caractères entre triple 'quote' (""" ou ''') qui sera considérée un commentaire.

```
def fonction(arg1, arg2) :
    """
    Je documente ma fonction.
    J'explique rapidement a quoi sert cette fonction.
    arg1 : J'ecris a quoi correspond cet argument.
    arg2 : J'ecris a quoi correspond cet argument.
    return : J'indique ce que retourne la fonction
    """
    pass
```

Voici un exemple d'une manière simple de documenter clairement vos fonctions. Il faut au minimum expliquer ce que fait la fonction. Une documentation complète détaille aussi le sens de chacun des arguments et ce que retourne la fonction. Avec une bonne documentation, personne ne doit avoir besoin de lire le code pour comprendre à quoi sert la fonction et ce qu'elle fait. Cela permet de la conserver en tant que boîte noire qui ne regarde que celui qui l'a implémentée.

Remarque : Le mot clé *pass* vous permet de sortir d'une structure. Il peut être utile par exemple lorsque vous développez un projet. Une bonne technique de développement étant de définir les fonctions puis l'algorithme principal et enfin d'implémenter les fonctions. Ce mot clé vous permet donc d'avoir toutes vos fonctions définies sans avoir à les implémenter à l'avance.

2.2 Passer plusieurs arguments d'une fonction à l'aide d'une séquence

Il est tout à fait possible de passer pour ce type d'argument une séquence, pour cela, il suffit de la faire précéder d'une étoile pour la convertir.

```
def fonction(*arguments) :
    """Test de fonction avec un nombre indefini d'arguments.
    arguments : Une sequence a ecrire en console."""
    for element in arguments :
        print(element)

fonction(*'abcdef')
```

Si vous n'utilisez pas l'opérateur '*', votre séquence sera alors considérée comme n'étant qu'un seul argument ! Cela est dû au typage dynamique. En effet, n'importe quel argument de votre fonction peut prendre n'importe quel type, il faut donc préciser ici que chaque élément de la séquence est un argument à part entière.

2.3 Nombre indéfini d'arguments

source : tuto de suzy

Il existe des fonctions qui acceptent un nombre indéterminé d'arguments, c'est le cas de la fonction print par exemple. Comment cela fonctionne-t-il ? Et bien, c'est assez simple. Pour cela, il est possible de le faire de deux manières : avec des arguments anonymes ou avec arguments associés à des clés. Nous allons donc voir les deux manières de créer des fonctions avec un nombre indéterminé de paramètres.

2.3.1 Arguments anonymes

La syntaxe est particulièrement simple. En fait, vous allez préciser une séquence (liste, tuple ou chaîne de caractères) comme argument. Le fait que cet argument doivent être une séquence se précise par l'utilisation de l'opérateur '*' juste avant le nom de l'argument :

```
1 def fonction(*arguments) :
2     """Test de fonction avec un nombre indefini d'arguments.
3     arguments : Une sequence a ecrire en console."""
4     for element in arguments :
5         print(element)
6
7 fonction(43, 38, "Peuh !", True)
```

De cette façon, il est donc possible d'appeler ma fonction avec un nombre indéfini d'arguments mais attention, cela est possible à une seule condition : la séquence d'arguments doit impérativement se situer en dernier argument de la fonction ! Les arguments s'écrivent donc dans l'ordre suivant dans la déclaration d'une fonction : les arguments obligatoires, les arguments avec valeur par défaut (facultatifs) et enfin des arguments supplémentaires si nécessaires.

En utilisant l'astuce [§ 2.2, p5] ça donne :

```
1 def fonction(*arguments) :
2     """Test de fonction avec un nombre indefini d'arguments.
3     arguments : Une sequence a ecrire en console."""
4     for element in arguments :
5         print(element)
6
7 fonction(*'abcdef')
```

2.3.2 Arguments avec clé

Si vous voulez associer une clé à chacun des arguments que vous donnez en supplément à une fonction, il va falloir utiliser un dictionnaire. Et ce coup-ci, c'est l'opérateur '**' qui sera utilisé.

```
1 def fonction(**arguments) :
2     """Test de fonction avec un nombre indefini d'arguments.
3     arguments : Un dictionnaire a ecrire en console."""
4     for cle in arguments :
5         print(cle,arguments[cle])
6
7 fonction(arg1 = 'Peuh !', arg2 = 38)
```

Vous verrez en exécutant ce code que l'on peut donc facilement créer des couples (clé,valeur) à passer en paramètre. La contrainte est la même que précédemment et le dictionnaire d'arguments doit être impérativement placé en dernier dans la liste d'arguments lors de la déclaration de la fonction. Encore une fois, pour convertir un dictionnaire non pas un seul argument mais en une suite d'arguments, il faut utiliser l'opérateur '**' devant le dictionnaire passé en argument.

```
1 def fonction(**arguments) :
2     """Test de fonction avec un nombre indefini d'arguments.
3     arguments : Un dictionnaire a ecrire en console."""
4     for cle in arguments :
5         print(cle,arguments[cle])
6
7 fonction(**dict(arg1 = 'Peuh !', arg2 = 38))
```

2.4 Importer des fonctions

On peut importer des fonctions très facilement. Il convient pourtant de prendre de bonnes habitudes dès le départ. Par exemple, pour importer la fonction `sin()`, on peut choisir une des lignes suivantes :

```
1 import math
2 from math import *
3 from math import sin
4 import math as m
```

Ceci est donné uniquement à titre d'exemple. La première ligne importe toute la librairie math, et pour appeler la fonction sinus, il faudra faire `math.sin(x)`

Dans le cas de la deuxième ligne, on pourra faire `sin(x)`

La 3e ligne est identique à la 2e à ceci près qu'elle n'importe que la fonction sinus, et pas les autres fonctions de la librairie math. Ça permet de gagner un peu de place en n'important que ce dont on a besoin.

La 4e et dernière ligne importe toute la librairie math et définit un alias à celle ci. Ainsi, pour appeler la fonction sinus, on fera `m.sin(x)`.

Maintenant, voici ce qu'il faut éviter de faire : Il ne faut pas utiliser les lignes 2 ou 3 qui nous font perdre l'information d'où provient la fonction qu'on utilise.

Personnellement, je continue à le faire pour la librairie math, mais c'est la seule exception. Par exemple pour les plots, j'utilise *pylab* via

```
1 import pylab as pl
```

Ce qui fait que quand j'utilise une fonction via `pl.nomdefonction()` je sais qu'elle provient de la librairie pylab.

L'autre avantage de cette méthode est qu'elle permet d'éviter les conflits entre deux fonctions qui auraient le même nom, et provenant de librairies différentes [1].

Enfin le dernier avantage, et non le moindre, est qu'en important via `import` *`module`* ou `import` *`module`* `as` *`alias`*, ne seront chargé en mémoire que les fonctions effectivement utilisées.

⚠ Avec `from` *`module`* `import *`, on importe absolument tout, y compris les variables qui pourraient être définies. C'est donc à éviter, sauf cas particulier.

3 Manipuler des fichiers

Pour manipuler un fichier, il faut dans un premier temps l'ouvrir (et spécifier si on veut pouvoir écrire, lire, ou écrire sans effacer le fichier au moment de l'ouverture), puis effectuer diverses opérations dans un second temps et enfin le fermer.

```
1 fichier = open("fichier.txt",'w')
2 fichier.write("ligne 1\nligne 2")
3 fichier.close()
```

`'w'` signifie que l'on souhaite écrire dans le fichier. Il signifie aussi que le fichier sera effacé lors de l'ouverture (au cas où le fichier existe). Si on ne souhaite pas effacer le fichier, il faut utiliser `'a'` qui va permettre d'écrire à la suite du fichier.

Par exemple, voici la fonction que j'utilise pour écrire un fichier log de mon script.

```
1 def writelog(texte_log)
2   log = open(nom_fichier_log,'a')
3   log.write("["+str(time.strftime('%d/%m/%Y'\
4  %H:%M:%S'))+"] "+texte_log+"\n")
5   log.close()
```

(cette fonction ne retourne rien vu qu'il n'y en a pas besoin. Au début, je mettais `return 0` mais pour l'instant je l'ai enlevé).

Pour lire un fichier, on procède de la manière suivante

```
1 fichier = open("fichier.txt",'r')
2 t = fichier.readlines()
3 fichier.close()
```

Le fichier t est une liste comprenant un élément pour chaque ligne du fichier lu. `fichier.readline()` ne lit qu'une seule ligne à la fois. `fichier.read(nb)` permet de lire `nb` caractères à partir de là où s'était arrêté la lecture.

1. Par exemple la librairie pylab définit d'autres fonction trigonométriques (`sin()`, ...)

Remarque : À noter que chaque ligne se termine par `\n`.

4 Modules

Pour les commandes basiques d'importations, se reporter à [§ 2.4, p6].

4.1 Importer des modules qui ne sont pas dans le répertoire courant.

En ajoutant

```
import sys
sys.path.append(monchemin)
```

on peut spécifier un nouveau chemin pour un répertoire contenant des modules. Ainsi, python vérifiera dans ce dossier si le module qu'on souhaite importer s'y trouve (en plus des autres répertoires définis automatiquement et du répertoire courant)

Depuis Python 2.5, il est également possible d'effectuer des importations relatives. Pour cela, il vous faut d'abord créer un fichier `__init__.py` dans le répertoire contenant les modules à importer. Ce fichier peut être vide. Par exemple, considérons un répertoire locale qui contient un répertoire `A` lui même contenant un fichier `__init__.py` vide et un fichier `testA.py`. Depuis votre répertoire local, vous pouvez importer `testA` par l'instruction `import A.testA`, `from A import testA` ou `from A.testA import *`.

4.2 subprocess

Pour lancer une commande dans un processus séparé on fait :

```
process = subprocess.Popen("ls")
```

Il y a en pratique deux techniques. Où bien on lance la commande et les arguments en séparant :

```
process = subprocess.Popen(["ls", "-l"])
```

ou bien on lance une chaîne de caractère, et dans ce cas, on spécifie que c'est ce qu'on veut exécuter dans le terminal

```
process = subprocess.Popen("ls -l", shell=True)
```

Pour récupérer la sortie de la commande, on y accès via

```
sortie = process.stdout.readlines()
```

à condition d'avoir lancé le script avec l'option `stdout=subprocess.PIPE`; dans le cas de l'exemple précédent, ça donne donc :

```
process = subprocess.Popen("ls -l", shell=True, stdout=subprocess.PIPE)
sortie = process.stdout.readlines()
print sortie
```

Remarque : `sortie` est donc un objet-fichier (la même chose que si on avait fait `sortie = open("fichier.txt", 'r')`).

Sinon, on peut écrire la sortie dans un fichier de la façon suivante :

```
sortie = open("sortie.txt", 'w')
process = subprocess.Popen("ls -l", shell=True, stdout=sortie)
sortie.close()
```

Pour que le script attende la fin du processus avant de continuer, on fait

```
fail = process.wait()
```

à quoi on peut rajouter un

```
if (fail==0):
  print "OK"
else:
  print "fail"
```

5 Tout savoir sur les listes

On définit une liste de la façon suivante :

```
a = [1.2, 2.3, 4.1, 5.7]
```

On peut appeler un élément à l'aide de crochets. On aura

```
a[0] = 1.2
a[-1] = a[3] = 5.7
a[-2] = a[2] = 4.1
```

On peut rajouter un élément à la liste via

```
a.append(6.8)
```

On aura alors

```
a = [1.2, 2.3, 4.1, 5.7, 6.8]
```

pour avoir le nombre d'éléments de la liste, il suffit de faire

```
nb = len(a)
```

Pour supprimer d'une liste les n premiers éléments et les m derniers, on fera

```
for indice in range(n):
  del(liste[0])

for indice in range(m):
  del(liste[-1])
```

Pour récupérer l'indice d'une certaine valeur dans une liste :

```
>>> a.index(2.3)
1
```

Il existe aussi les fonctions *map* et *zip* qui ont le même principe que sous Maple.
La fonction *zip* retourne une liste de tuples

```
>>> print zip(['a', 'b', 'c'], [1, 2, 3])
[''a', a), ('b', 2), ('c', 3)]
```

Remarque : Ça marche aussi avec plusieurs séquences. Les chaînes de caractères sont considérées comme des séquences, on peut donc les zipper aussi.
À noter que la sortie de la commande sera de la même longueur que la liste la plus courte passée en arguments.

La fonction *map* applique une fonction à chaque élément de la séquence passée en argument.

```
>>> print map(abs, [-5, -7, -12])
[5, 7, 12]
```

qui est l'équivalent de

```
>>> print [abs(i) for i in [-5, -7, -12]]
[5, 7, 12]
```

à part le fait que *map* est plus rapide.
Il est possible d'utiliser sa propre fonction. Il est possible d'utiliser une fonction qui nécessite plusieurs arguments, et de passer plusieurs listes à la fonction *map*.

```
>>> print map(max, [4, 5, 6], [1, 2, 9])
[4, 5, 9]
```

Il est possible de convertir une liste en tuple via la commande `tuple`

```
>>> a = [1, 2, 3]
>>> tuple(a)
(1, 2, 3)
```

5.1 Sélectionner une partie d'une liste

Il est possible de prendre une partie de liste :

```
>>> t = range(5)
>>> t
[0, 1, 2, 3, 4]
>>> t[1:3]
[1, 2]
```

⚠ À partir d'une liste d'indice `[i:j]`, python retourne les éléments allant de i à $j-1$. Ainsi pour une liste de n éléments, aux indices allant de 0 à $n-1$, pour avoir la liste complète en spécifiant les indices il faut faire :

```
>>> liste[0:n]
[0, ..., n-1]
```

5.2 List-comprehension

Cette technique permet d'agir sur les listes très simplement et très rapidement. Dans bien des cas (pour ne pas dire tous), cette méthode sera plus rapide que l'équivalent avec une boucle :

```
list2 = [element * 2 for element in list]
list3 = [element + 3 for element in list if element>3]
```

5.3 Array et Numpy

```
import numpy as np
```

Permet de multiplier terme à terme des matrices de manière très simple. On définit une matrice paragraph

```
m = np.array([[1, 2, 3], [4, 5, 6]])
```

Après avoir galéré pendant plusieurs heures pour créer et remplir une matrice au fur et à mesure, j'ai trouvé plus pratique de définir une liste comme on le souhaite, avec la souplesse que l'on connait de Python, puis convertir cette liste en array pour faire les calculs.

Dans la suite, on considère une matrice `m`. On effectue la somme de tous les éléments de la matrice par

```
>>> m.sum()
21
```

Pour effectuer la somme des éléments colonne par colonne (on somme donc sur l'indice des lignes, c'est à dire le premier indice) :

```
>>> m.sum(axis=0)
array([5, 7, 9])
```

La même chose ligne par ligne (on somme donc sur les colonnes avec le numéro de ligne fixé) :

```
>>> m.sum(axis=1)
array([ 6, 15])
```

Remarque : Apparemment, il existe `np.append(array, truc-à-rajouter, options)` qui permet de rajouter un truc à un array, et notamment de spécifier si on veut en ligne ou en colonne. Le truc c'est qu'apparemment, il reconstruit la matrice à chaque fois, donc pas sur qu'on gagne en temps d'exécution.

`np.zeros()` et `np.empty()` permettent de définir des matrices en spécifiant leur taille

```
np.zeros((2,3))
```

Il est possible (et c'est très pratique) de définir une matrice de la même taille qu'une autre, en spécifiant qu'on ne souhaite que des 0 ou une matrice vide. Pour celà, on utilise

```
np.zeros_like(m)
np.empty_like(m)
```

5.3.1 Manipulation des array

On peut sélectionner les valeurs répondant à un ou plusieurs critères avec :

```
a[a<2.]
a[(a<10) & (a>2)]
```

Pour modifier des sous listes, il faut faire attention à ce qu'on manipule.

```
a[a<2.] = a * 10
```

ne va pas multiplier tous les éléments inférieur à 2. par 10 car les indices du sous tableaux se voient affecter la multiplication par 10 de l'indice correspondant du tableau non modifié. Ainsi le 2e élément du sous tableau vaudra 10 fois le 2e élément du tableau complet !

Pour faire cette opération il faut faire :

```
too_low = a < 2.
a[too_low] = a[too_low] * 10
```

Ces tests suppriment automatiquement les valeurs 'nan' qui ne vérifient aucun test. Si on veut juste supprimer les 'nan', on peut faire :

```
a[np.isfinite(a)]
```

6 Programmation objet

6.1 Les fonctions sont des objets

Voici une astuce pour choisir une fonction en fonction de la valeur d'un paramètre, ceci de façon rapide et lisible :

```
def function1():
    print 'You chose one.'
def  function2():
    print 'You chose two.'
def  function3():
    print 'You chose three.'
#
# switch est notre dictionnaire de functions
switch = {
    'one': function1,
    'two': function2,
    'three': function3,
    }
#
# le choix peut etre 'one', 'two', or 'three'
choice = raw_input('Enter one, two, or three :')
#
# call one of the functions
try:
    result = switch[choice]
except KeyError:
    print 'I didn\'t understand your choice.'
else:
    result()
```

`result = switch[choice]` est la ligne qui réalise le gros du travail. `switch[choice]` renvoie un de nos objets fonctions (ou lève une `KeyError`). Ensuite nous exécutons `result()` qui l'appelle.

On peut économiser quelques lignes de code via :

```
# appelle une des fonctions
try:
    switch[choice]()
except KeyError:
    print 'I didn\'t understand your choice.'
```

7 Graphiques avec pylab

J'utilise le module *pylab* que j'importe par

```
import pylab as pl
```

Le principe est le même que pour les graphiques avec matlab. On définit des figures :

```
pl.figure(1)
```

Et on peut ensuite y afficher des graphiques, les exporter, modifier les propriétés etc...

Remarque : Si on n'utilise qu'une seule figure, il n'est pas nécessaire de définir son nombre, qui n'est utile que pour pouvoir changer de figure, nettoyer une figure pour la remettre à zéro et afficher autre chose à la place. Par défaut, le numéro est **1**.

Pour remettre une figure à zéro :

```
pl.figure(1) # definit la figure 1 comme figure courante
pl.clf() # efface la figure courante (clf pour clear figure)
```

7.1 Afficher un graphique

Pour faire un plot

```
pl.plot(x,y)
```

et l'afficher à l'écran

```
pl.show()
```

7.2 Exporter un graphique

Pour exporter la figure courante, il suffit de faire :

```
pl.savefig('ma_figure.png')
pl.savefig('ma_figure.svg')
pl.savefig('ma_figure.pdf')
```

7.3 Plusieurs graphiques sur une même page : subplots

subplot pylab fonctionne comme les plots dans *matlab*. Il existe aussi la possibilité de faire des subplots.

```
pl.figure(1)

# en haut a gauche
pl.subplot(221)
pl.plot(x,y)

#en haut a droite
pl.subplot(222)
pl.plot(x,y)

#en bas a gauche
pl.subplot(223)
pl.plot(x,y)

#en bas a droite
pl.subplot(224)
pl.plot(x,y)

pl.show()
```

7.4 Personnalisations diverses

Pour aller un peu plus loin

```
pl.figure(1) # Pour specifier que la figure active est la 1
pl.clf() # pour remettre a zero la figure active
pl.grid(True)
pl.xlabel("abscisse")
pl.ylabel("ordonnee")
pl.plot(x,y, 'r.', label="y=f(x)")
pl.plot(x,z, 'b-', label="z=g(x)")
pl.legend() # afficher la legende
pl.show()
```

7.5 Utilisation de symbole au lieu de ligne

```
pl.plot(x,z, 'o', markersize=5)
```

On peut utiliser des cercles `o`, des croix `x` ou des carrés `s`

7.6 Graphiques logarithmiques

À la place de `plot` on peut aussi faire :

```
pl.semilogx(x,y)
pl.semilogy(x,y)
pl.loglog(x,y)
```

7.7 Zoom et étendue des axes

Pour spécifier le zoom sur la figure, il faut rajouter :

```
pl.axis([xmin, xmax, ymin, ymax])
```

Python ne va pas forcément zoomer sur les limites du graphique. Je veux dire par là que si la valeur maximale en abscisse est 3, il affichera peut-être des limites jusqu'à 5. Pour forcer à avoir une zone graphique qui affiche nos courbes et uniquement nos courbes, il faut utiliser :

```
pl.axis('tight')
```

et comme je suis pointilleux et jamais content, j'aime bien que les axes commencent à zéro, ce qu'on peut faire via :

```
pl.axis('tight')
pl.xlim(xmin=0)
pl.ylim(ymin=0)
```

Les commandes **xlim** et **ylim** peuvent spécifier respectivement les limites (**xmin**, **xmax**, **ymin**, **ymax**) des axes. En ne spécifiant qu'une valeur, l'autre est laissée inchangée.

7.8 Histogrammes

Pour afficher un histogramme :

```
pl.hist(x, normed=True, bins=20)
```

Pour afficher plusieurs histogrammes à la fois :

```
pl.hist([x,y], normed=True, bins=20, label=['x', 'y'])
```

On peut changer le style avec `histtype='style'` où `style` peut prendre les valeurs suivantes :

- bar
- barstacked
- step
- stepfilled

D'autres options pratiques sont :

```
cumulative=True
normed=True
bins=20
```


Quand on veut afficher plusieurs histogrammes dans le même plots en plusieurs fois, il faut spécifier un range de bins bien définis. J'ai déjà fait deux groupes de deux histogrammes, et j'avais de gros soucis dans les valeurs, ceci dû à ces bins qui affichaient les mêmes valeurs mais qui au final ne correspondaient pas aux mêmes choses.

7.9 Afficher une carte de valeurs (du 3D avec le z représenté par un code de couleur)

On peut afficher une matrice et spécifier des axes (par défaut, il n'affiche que les indices de la matrice, mais on peut associer des plages de valeurs arbitraires aux axes x et y)

```
liste_x = range(5)
liste_y = range(5)
pl.figure(1)
pl.xlabel("axe des x")
pl.ylabel("axe des y")
import pylab as pl
pl.matshow(matrice, fignum=1)
pl.colorbar()
axes = pl.gca()
axes.set_xticks(range(len(liste_x)))
axes.set_xticklabels(liste_x)
axes.set_yticks(range(len(liste_y)))
axes.set_yticklabels(liste_y)
pl.show()
```

8 Faire des statistiques

8.1 Bases

Dans *numpy*, il existe les fonctions statistiques de bases, notamment la *valeur moyenne* et l'*écart-type* :

```
a = range(10)
np.mean(a)
np.std(a)
```

8.2 Test de Kolmogorov-Smirnov

Pour celà, il faut utiliser le module suivant :

```
import scipy.stats.stats as scss
```

et on appelle le test via

```
(d_value, p_value) = scss.ks_2samp(a,b)
```

où a et b sont deux échantillons de données (des listes ou des array quoi).

9 Exécuter des commandes système

Le module *os* permet d'utiliser plusieurs commandes systèmes dont voici quelques équivalents

commande bash : équivalent python
cd 'dossier' : os.chdir('nom dossier')
mkdir : os.mkdir('dossier')
ls : os.listdir('.')
pwd : os.getcwd()
rm 'fichier.txt' : os.remove('fichier.txt')

Une commande permet de manière plus générale de lancer une commande système :

```
os.system('cmd argument')
```

Par exemple, on peut très bien faire les commandes précédentes via la commande system

```
# methode 1
os.system('cd \'dossier\'')
# methode 2
os.system("cd 'dossier'")

os.system("mkdir 'dossier'")
os.system("ls")
os.system("pwd")
os.system("rm 'fichier.txt'")
```

Dans ce même module, il existe une commande très pratique pour tester l'existence d'un fichier ou dossier (pratique pour tester avant suppression ou changement de répertoire, ça évite les bugs du programme)

```
nom_fichier = "simulation.log"
if os.path.exists(nom_fichier):
  os.remove(nom_fichier)
```

Si, par contre, on désire stocker le résultat d'une commande, on ne peut utiliser `system`. On doit utiliser un autre module, le module *commands* et utiliser la commande suivante

```
liste = commands.getoutput("ls")
# facultatif mais utile
liste = liste.split("\n")
```

 Préférer les méthodes décrites dans [§ 4.2, p8] plutôt que les méthodes du module *os*. Je sais plus pourquoi, mais parait que c'est mieux.

10 Erreurs

10.1 Unindent

```
IndentationError: unexpected indent
```

C'est normalement dû au fait qu'il y a dans le code source des indentations avec tabulation et espaces. Un simple rechercher/remplacer permet de corriger l'erreur normalement.

11 Astuces

11.1 Copier une liste (et non faire un lien virtuel)

Par défaut, en python, si vous assignez une variable contenant une liste à une autre variable, la 2e variable sera un lien virtuel vers la première. La conséquence la plus importante est que si vous modifiez la 2e variable, vous modifiez également la première.

Exemple :

```
>>> t1 = range(5)
>>> t2 = t1
>>> print t1, t2
[0, 1, 2, 3, 4] [0, 1, 2, 3, 4]
>>> del(t2[2])
>>> print t1, t2
[0, 1, 3, 4] [0, 1, 3, 4]
```

Afin d'éviter ce genre de problèmes, on peut faire les choses suivantes :

```
t2 = list(t1)
```

ou

```
t2 = t1[:]
```

11.2 Tester si une variable existe

```
try:
  print toto
except NameError:
  print "toto n'existe pas"
```

ou alors

utiliser la méthode `has_key()` des dictionnaires internes répertoriant les variables. Si on veut tester l'existence de la variable x qui est une variable locale ou globale, il faut utiliser respectivement

```
vars().has_key('x')
globals().has_key('x')
```

ou encore

```
if ('x' in locals()):
  print("'x' est defini")
```

11.3 Tester le type d'une variable

```
>>> liste=['test', 'test']
>>> if isinstance(liste, type([])):
...  print "c'est une liste"
"c'est une liste"
```

On peut aussi faire

```
>>> liste=['test', 'test']
>>> if isinstance(liste, list):
...  print "c'est une liste"
"c'est une liste"
```

On peut aussi tester plusieurs types à la fois

```
>>> liste=['test', 'test']
>>> if isinstance(liste, (list,int)):
...  print "c'est une liste ou un entier"
"c'est une liste"
```

Sinon, plus simple :

```
>>> liste=['test', 'test']
>>> if (type(liste)==list):
...  print "c'est une liste"
"c'est une liste"
```

Pour connaître le type à rentrer pour le test, il suffit de faire

```
>>> liste=['test', 'test']
>>> type(liste)
<type 'list'>
```

Le type à rentrer est simplement la valeur entre guillemets, donc ici `list`

11.4 Arrondir les floats

Il est possible d'arrondir les floats à l'aide de

```
>>> test = 1.12456124
>>> round(test,3)
1.12499999
```

On pourrait croire que ceci ne marche pas. En fait, c'est dû au fait que le processeur arrondi à la valeur la plus proche qu'il connait (penser au fait qu'un ordinateur ne connait pas les réels par défaut).

Si on ne veut que quelques chiffres significatifs, c'est généralement pour un affichage. On n'a alors aucun problème, car lors de la conversion en string du flottant arrondi, il restera les bons chiffres significatifs (à condition qu'on ait fait un round avant, ça va de soi)

```
>>> test = 1.12456124
>>> test2 = round(test,3)
>>> str(test2)
'1.125'
```

11.5 Lister un type de fichier

En console, il est possible de lister des types de fichier en utilisant les expressions régulières, comme par exemple

```
ls *.txt
ls test*
```

Avec python, si on souhaite faire ça dans un processus, il faudra utiliser

```
process = subprocess.Popen("ls *.txt", shell=True)
```

En effet, `process = subprocess.Popen(["ls", "*.txt"])` renverra une erreur. Ne me demandez pas pourquoi, j'en sais rien.

Remarque : Pour récupérer la sortie, il faut bien évidemment faire comme expliqué dans la section [§ 4.2, p8].

11.6 Modifier la ligne précédemment affichée dans le terminal (progression)

```
import sys, time

for i in range(1, 6):
    sys.stdout.write (str(i) + chr(13))
    sys.stdout.flush()
    time.sleep (1)
```

Par défaut quand on fait un **print** en console, le **print** envoi en fin de ligne un `chr(13)` (Carriage Return ; CR) et un `chr(10)`, (Line Feed ; LF).

CR (`chr(13)`) renvoi en début de ligne et LF effectue un saut de ligne. La combinaison de ces deux caractères équivaut à un renvoi à la ligne (qui sous GNU/Linux se code CR+LF).

En écrivant uniquement `chr(13)` (CR), je retourne au début de la ligne sans faire un saut de ligne, ce qui permet de réécrire sur la ligne précédente. C'est pratique quand il y a des affichage textuels de barres de progression par exemple.

11.7 Personnaliser Python

Cette astuce doit surement fonctionner avec d'autres language, je n'ai utilisé que python, donc à vous de voir comment généraliser.

Je souhaitais pouvoir exécuter certains scripts python depuis n'importe quel répertoire sans avoir à entrer le chemin complet. je souhaitais aussi pouvoir importer les modules que j'ai créé sans devoir modifier le `sys.path` à chaque fois et dans chaque script.

Afin de pouvoir exécuter les scripts depuis n'importe quel dossier, il faut déjà regrouper tous ces scripts dans un dossier particulier. Par exemple, j'ai mis dans mon `/home` un dossier `python/traitement_simu` où je stocke les scripts que je veux exécuter pour chaque simulation, et un dossier `python/modules` où je stocke tous mes modules.

Les modifications sont à effectuer dans `.bash_profile`.

Il faut rajouter le dossier `python/traitement_simu` dans la variable PATH et il faut créer une variable PYTHONPATH.

⚠ Afin que ceci fonctionne, il faut *exporter* ces variables.

En supposant que votre fichier `.bash_profile` est vide (ou n'existe pas.[2]), voici ce qu'il faut rajouter :

```
PATH=$PATH:$HOME/python/traitement_simu
PYTHONPATH=$HOME/python/modules

export PATH PYTHONPATH
```

2. Ce fichier est à placer dans votre `/home`

11.8 récupérer le nom ou l'extension d'un fichier

Il existe une méthode dans le module *os*

```
ext = os.path.splitext(file_name)[1]
name = os.path.splitext(file_name)[0]
```

ou

```
(name, ext) = os.path.splitext(file_name)
```

11.9 Optimisations

(J'ai eu la flemme de traduire, mais l'idée y est)

If you feel the need for speed, go for built-in functions - you can't beat a loop written in C. Check the library manual for a built-in function that does what you want. If there isn't one, here are some guidelines for loop optimization :

- Rule number one : only optimize when there is a proven speed bottleneck. Only optimize the innermost loop. (This rule is independent of Python, but it doesn't hurt repeating it, since it can save a lot of work. :-)
- Small is beautiful. Given Python's hefty charges for bytecode instructions and variable look-up, it rarely pays off to add extra tests to save a little bit of work.
- Use intrinsic operations. An implied loop in map() is faster than an explicit for loop ; a while loop with an explicit loop counter is even slower.
- Avoid calling functions written in Python in your inner loop. This includes lambdas. In-lining the inner loop can save a lot of time.
- Local variables are faster than globals ; if you use a global constant in a loop, copy it to a local variable before the loop. And in Python, function names (global or built-in) are also global constants !
- Try to use map(), filter() or reduce() to replace an explicit for loop, but only if you can use a built-in function : map with a built-in function beats for loop, but a for loop with in-line code beats map with a lambda function !
- Check your algorithms for quadratic behavior. But notice that a more complex algorithm only pays off for large N - for small N, the complexity doesn't pay off. In our case, 256 turned out to be small enough that the simpler version was still a tad faster. Your mileage may vary - this is worth investigating.
- And last but not least : collect data. Python's excellent profile module can quickly show the bottleneck in your code. if you're considering different versions of an algorithm, test it in a tight loop using the time.clock() function.

12 Avancé

12.1 Passer des arguments à un script

Il y a un moyen très simple de passer des arguments à un script python en ligne de commande. Il suffit de faire par exemple :

```
> ./test.py 1 "hey"
```

Le script alors contenir les commandes suivantes :

```
import sys

print sys.argv[1] # affiche arg1
print sys.argv[2] # affiche arg2
```

Les arguments sont des chaînes de caractères, à vous de les convertir dans le format qui vous convient en fonction de l'ordre.

Remarque : À noter que le premier élément de la liste **sys.argv** est le chemin absolu vers le script

```
(Pdb) print sys.argv
['/home/login/scripts/test.py', '1', 'hey']
```

Il est possible de faire des choses beaucoup plus compliquées en mettant des options au script du style `-d` ou `-debut` mais c'est beaucoup plus compliqué. Il faut pour cela importer le module **getopt**. Voici un bout de code que j'avais fait en utilisant ce module :

```
import sys, getopt

try:
    opts, args = getopt.getopt(sys.argv[1:],
    "hdf", ["help", "debut=", "fin="])
except getopt.GetoptError:
    print("les options sont h, d et f")
    sys.exit(2)

for opt, arg in opts:
    if opt in ("-h", "--help"):
        print("les options sont h, d et f")
        sys.exit()
    elif opt in ("-d", "--debut"):
        try:
            indice_simu_debut = int(arg)
        except:
            print("L'argument n'est pas un entier")
    elif opt in ("-f", "--fin"):
        try:
            indice_simu_fin = int(arg)
        except:
            print("L'argument n'est pas un entier")

autres_arguments = " ".join(args)
```

On peut remarquer que chaque option a un format court et un format long.

Index