Python - cours intensif pour les scientifiques

Partie 3 : Python avancé

Par Rick Muller - Raphaël Seban (traducteur)

Date de publication : 10 août 2014

Pour de nombreux scientifiques, Python est LE langage de programmation par excellence, car il offre de grandes possibilités en analyse et modélisation de données scientifiques avec relativement peu de charge de travail en termes d'apprentissage, d'installation ou de temps de développement. C'est un langage que vous pouvez intégrer en un week-end, puis utiliser une vie durant.

Retrouvez les deux premières parties de cet article :

- **Partie 1 : Présentation de Python**
- **Partie 2 : Modules NumPy et SciPy**

Les commentaires et les suggestions d'amélioration sont les bienvenus, alors, après votre lecture, n'hésitez pas. **Commentez**

En complément sur Developpez.com

- Condensé Python pour les scientifiques - Partie 1
- Condensé Python pour les scientifiques - Partie 2
- Tutoriel Matplotlib

I - Introduction....3
II - Analyse de la sortie console....3
III - Aller plus loin avec le formatage et les traitements des chaînes de caractères....9
IV - Arguments facultatifs....14
V - Listes en compréhension et générateurs....18
VI - Fonctions de fonctions (fermetures)....20
VII - La sérialisation : conserver pour plus tard....21
VIII - Programmation fonctionnelle....22
IX - Programmation Orientée Objet (POO)....23
X - Remerciements....28
XI - Remerciements Developpez....29

I - Introduction

IP[y]: Notebook

*Cette partie est consultable également au format « notebook » en suivant le lien : **Notes : Partie 3 - Python avancé**.*

II - Analyse de la sortie console

Alors que de plus en plus de travail est effectué grâce aux ordinateurs, nous générons de plus en plus de sorties console - généralement dans des fichiers texte - que nous devons ensuite analyser et puis, presque immanquablement, inscrire nos résultats dans un autre fichier.

Supposons que nous ayons la sortie suivante :

In :
```
myoutput = """\
@ Step        Energy      Delta E   Gmax     Grms     Xrms     Xmax   Walltime
@ ---- ---------------- -------- -------- -------- -------- -------- --------
@    0   -6095.12544083  0.0D+00  0.03686  0.00936  0.00000  0.00000   1391.5
@    1   -6095.25762870 -1.3D-01  0.00732  0.00168  0.32456  0.84140  10468.0
@    2   -6095.26325979 -5.6D-03  0.00233  0.00056  0.06294  0.14009  11963.5
@    3   -6095.26428124 -1.0D-03  0.00109  0.00024  0.03245  0.10269  13331.9
@    4   -6095.26463203 -3.5D-04  0.00057  0.00013  0.02737  0.09112  14710.8
@    5   -6095.26477615 -1.4D-04  0.00043  0.00009  0.02259  0.08615  20211.1
@    6   -6095.26482624 -5.0D-05  0.00015  0.00002  0.00831  0.03147  21726.1
@    7   -6095.26483584 -9.6D-06  0.00021  0.00004  0.01473  0.05265  24890.5
@    8   -6095.26484405 -8.2D-06  0.00005  0.00001  0.00555  0.01929  26448.7
@    9   -6095.26484599 -1.9D-06  0.00003  0.00001  0.00164  0.00564  27258.1
@   10   -6095.26484676 -7.7D-07  0.00003  0.00001  0.00161  0.00553  28155.3
@   11   -6095.26484693 -1.8D-07  0.00002  0.00000  0.00054  0.00151  28981.7
@   11   -6095.26484693 -1.8D-07  0.00002  0.00000  0.00054  0.00151  28981.7"""
```

Ces données proviennent d'une optimisation géométrique d'un cluster de la Silicon utilisant la suite logicielle de chimie quantique **NWChem™**. À chaque étape, le programme calcule l'énergie propre à la géométrie de la molécule, puis modifie la géométrie pour ajuster les calculs jusqu'à ce que l'énergie converge. J'ai obtenu cette sortie console grâce à la commande shell UNIX® :

```
% grep @ nwchem.out
```

Nous procédons ainsi, car NWChem fait précéder toutes ses lignes de résultat du signe arobase (@).

Nous pourrions effectuer toute l'analyse en Python - nous verrons comment tout à l'heure - mais attachons-nous tout d'abord à transformer ces données en un objet Python exploitable pour les graphiques.

Vous noterez que les données entrées dans la variable **myoutput** sont multilignes. Python permet en effet la notation d'une chaîne de caractères (*str*) sur plusieurs lignes grâce aux guillemets anglais triples (""") ou aux guillemets simples triples (''') qui délimitent de part et d'autre une chaîne de caractères comprenant toute sorte de caractères spéciaux tels que retour chariot (\n), tabulation (\t), etc. C'est une méthode bien pratique pour insérer l'équivalent du contenu d'un fichier directement dans une variable chaîne de caractères sans avoir à gérer le fichier lui-même dans le code, ligne à ligne, une ligne après l'autre. On pourra par la suite accéder à la variable et la traiter comme un seul gros paquet de données.

Nous commencerons par fractionner la chaîne de caractères contenue dans la variable **myoutput** en une liste Python (*list* ou *[]*) de plusieurs chaînes de caractères, chacune correspondant à une ligne de données affichée. Utilisons

la méthode splitlines() incluse dans notre variable chaîne de caractères (voir **documentation officielle**) pour la fragmenter en liste Python de chaînes de caractères à chaque fois qu'elle rencontre le caractère retour chariot (\n) :

*NdT : en programmation, lorsqu'une fonction est **incluse dans un objet**, on la nomme **méthode** pour la distinguer d'une fonction indépendante.*

In :
```
lines = myoutput.splitlines()
lines
```

Out :
```
['@ Step        Energy      Delta E   Gmax     Grms     Xrms     Xmax    Walltime',
 '@ ---- ---------------- -------- -------- -------- -------- -------- --------',
 '@    0   -6095.12544083  0.0D+00  0.03686  0.00936  0.00000  0.00000   1391.5',
 '@    1   -6095.25762870 -1.3D-01  0.00732  0.00168  0.32456  0.84140  10468.0',
 '@    2   -6095.26325979 -5.6D-03  0.00233  0.00056  0.06294  0.14009  11963.5',
 '@    3   -6095.26428124 -1.0D-03  0.00109  0.00024  0.03245  0.10269  13331.9',
 '@    4   -6095.26463203 -3.5D-04  0.00057  0.00013  0.02737  0.09112  14710.8',
 '@    5   -6095.26477615 -1.4D-04  0.00043  0.00009  0.02259  0.08615  20211.1',
 '@    6   -6095.26482624 -5.0D-05  0.00015  0.00002  0.00831  0.03147  21726.1',
 '@    7   -6095.26483584 -9.6D-06  0.00021  0.00004  0.01473  0.05265  24890.5',
 '@    8   -6095.26484405 -8.2D-06  0.00005  0.00001  0.00555  0.01929  26448.7',
 '@    9   -6095.26484599 -1.9D-06  0.00003  0.00001  0.00164  0.00564  27258.1',
 '@   10   -6095.26484676 -7.7D-07  0.00003  0.00001  0.00161  0.00553  28155.3',
 '@   11   -6095.26484693 -1.8D-07  0.00002  0.00000  0.00054  0.00151  28981.7',
 '@   11   -6095.26484693 -1.8D-07  0.00002  0.00000  0.00054  0.00151  28981.7']
```

Le fractionnement est une technique très répandue en traitement de texte. Nous venons d'utiliser la méthode splitlines() et nous utiliserons ensuite la méthode générique split() (voir **documentation officielle**) pour fractionner chaque ligne de texte en portions délimitées par une espace blanche.

À présent, nous voudrions faire trois choses :

- ignorer les lignes qui ne contiennent pas d'information utile ;
- fractionner les lignes intéressantes pour en extraire les portions qui nous intéressent ;
- traduire les données de sorte que l'on puisse les tracer sur un graphique.

Dans le bloc de données ici présent, nous ne sommes intéressés que par les colonnes **Energy**, **Gmax** (Gradient maximum à chaque étape) et **Walltime**.

Comme les données sont actuellement fractionnées en une liste Python de lignes de texte, nous pouvons les parcourir :

In :
```
for line in lines[2:]:
    # faire quelque chose avec chaque ligne
    words = line.split()
```

Examinons pas à pas : tout d'abord, on utilise une boucle d'itération for … in … pour parcourir chaque ligne de texte contenue dans **lines[2:]** que l'on place dans une variable d'itération **line**. Vous noterez que nous avons ignoré les lignes d'indices **lines[0]** et **lines[1]** dans la foulée de notre déclaration de boucle for … in … , car la première ligne contient les en-têtes de colonnes et la seconde des tirets de séparation.

Ensuite, on fractionne chaque ligne en portions délimitées par une espace blanche grâce à la méthode split(). Voici comment cette méthode fonctionne :

In :
```
import string
help(string.split)
```

Out :
```
Help on function split in module string:

split(s, sep=None, maxsplit=-1)
    split(s [,sep [,maxsplit]]) -> list of strings

    Return a list of the words in the string s, using sep as the
    delimiter string.  If maxsplit is given, splits at no more than
    maxsplit places (resulting in at most maxsplit+1 words).  If sep
    is not specified or is None, any whitespace string is a separator.

    (split and splitfields are synonymous)
```

En appelant split() sans aucun argument, on s'appuie sur le comportement par défaut de cette méthode tel que décrit ci-dessus. Comme on ne spécifie pas le séparateur à utiliser lors du fractionnement, la méthode utilise le séparateur par défaut, à savoir une espace blanche. Voyons ce que cela donne sur l'une de nos lignes :

In :
```
lines[2].split()
```

Out :
```
['@',
 '0',
 '-6095.12544083',
 '0.0D+00',
 '0.03686',
 '0.00936',
 '0.00000',
 '0.00000',
 '1391.5']
```

C'est presque exactement ce que nous voulons. Il ne nous reste plus qu'à récupérer les valeurs des colonnes **Energy** (index 2), **Gmax** (index 4) et **Walltime** (index 8) :

In :
```
for line in lines[2:]:
    # faire quelque chose avec chaque ligne
    columns = line.split()
    energy = columns[2]
    gmax = columns[4]
    time = columns[8]
    print energy,gmax,time
```

Out :
```
-6095.12544083 0.03686 1391.5
-6095.25762870 0.00732 10468.0
-6095.26325979 0.00233 11963.5
-6095.26428124 0.00109 13331.9
-6095.26463203 0.00057 14710.8
-6095.26477615 0.00043 20211.1
-6095.26482624 0.00015 21726.1
-6095.26483584 0.00021 24890.5
-6095.26484405 0.00005 26448.7
-6095.26484599 0.00003 27258.1
-6095.26484676 0.00003 28155.3
-6095.26484693 0.00002 28981.7
-6095.26484693 0.00002 28981.7
```

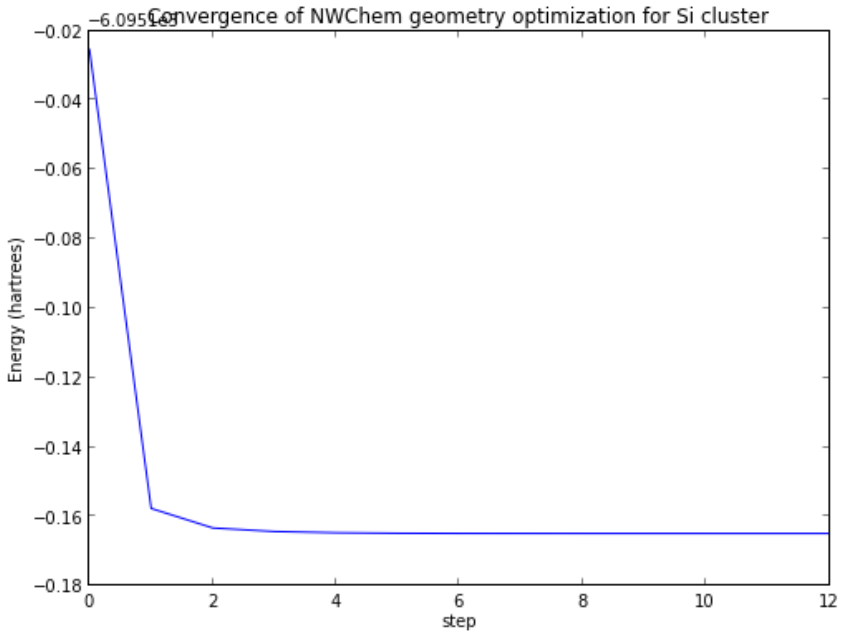
C'est bien pratique pour afficher à la console, mais si nous voulons nous servir de ces données pour effectuer des calculs ou pour tracer des graphiques, nous devons les convertir en nombres à virgule flottante, parce que pour le moment, ce sont toujours des chaînes de caractères. Utilisons la fonction objet float() pour convertir des chaînes de caractères en nombres à virgule flottante double précision. Nous aurons aussi besoin de conserver nos données dans une structure. Je ferais tout cela comme suit :

In :
```
data = []
for line in lines[2:]:
    # faire quelque chose avec chaque ligne
    columns = line.split()
    energy = columns[2]
    gmax = columns[4]
    time = columns[8]
    data.append((energy,gmax,time))
data = array(data)
```

À présent que nous avons nos données dans un tableau numpy, choisissons les colonnes à tracer :

In :
```
plot(data[:,0])
xlabel('step')
ylabel('Energy (hartrees)')
title('Convergence of NWChem geometry optimization for Si cluster')
```

<matplotlib.text.Text at 0x11116cf90>



J'écrirais ce code plus succinctement si je le faisais pour moi, mais c'est là un bout de code que l'on retrouve fréquemment écrit ainsi.

Supposons un instant que nos données sont fournies au format **CSV** (*Comma-Separated Values*), un format utilisé par Microsoft® Excel™ à l'origine et qui est de plus en plus utilisé comme format d'échange simple dans les applications de *big data*. Comment traiterions-nous ces données ?

In :
```
csv = """\
-6095.12544083, 0.03686, 1391.5
-6095.25762870, 0.00732, 10468.0
-6095.26325979, 0.00233, 11963.5
-6095.26428124, 0.00109, 13331.9
-6095.26463203, 0.00057, 14710.8
-6095.26477615, 0.00043, 20211.1
-6095.26482624, 0.00015, 21726.1
-6095.26483584, 0.00021, 24890.5
-6095.26484405, 0.00005, 26448.7
-6095.26484599, 0.00003, 27258.1
```

In :
```
-6095.26484676, 0.00003, 28155.3
-6095.26484693, 0.00002, 28981.7
-6095.26484693, 0.00002, 28981.7"""
```

Nous pouvons faire à peu près la même chose que tout à l'heure :

In :
```
data = []
for line in csv.splitlines():
    columns = line.split(',')
    data.append(map(float, columns))
data = array(data)
```

Deux différences significatives, toutefois. D'abord, je passe le caractère virgule (',') en argument de fonction split(), de sorte que la ligne est fractionnée en portions à chaque virgule rencontrée. Ensuite, pour simplifier grandement les choses, j'utilise la fonction objet map() qui me permet d'appliquer itérativement la fonction float() à chaque élément d'une liste Python (*list* ou *[]*) et qui retourne le résultat final dans une autre liste Python (*list* ou *[]*).

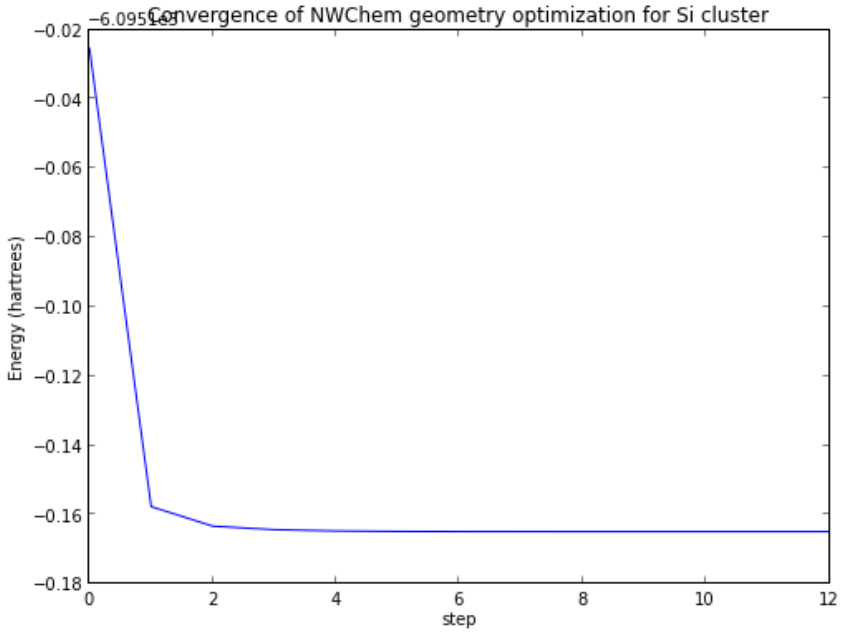
In :
```
help(map)
```

Out :
```
Help on built-in function map in module __builtin__:

map(...)
    map(function, sequence[, sequence, ...]) -> list

    Return a list of the results of applying the function to the items of
    the argument sequence(s).  If more than one sequence is given, the
    function is called with an argument list consisting of the corresponding
    item of each sequence, substituting None for missing values when not all
    sequences have the same length.  If the function is None, return a list of
    the items of the sequence (or a list of tuples if more than one sequence).
```

Malgré ces différences, le tracé final devrait être le même :

In :
```
plot(data[:,0])
xlabel('step')
ylabel('Energy (hartrees)')
title('Convergence of NWChem geometry optimization for Si cluster')
```

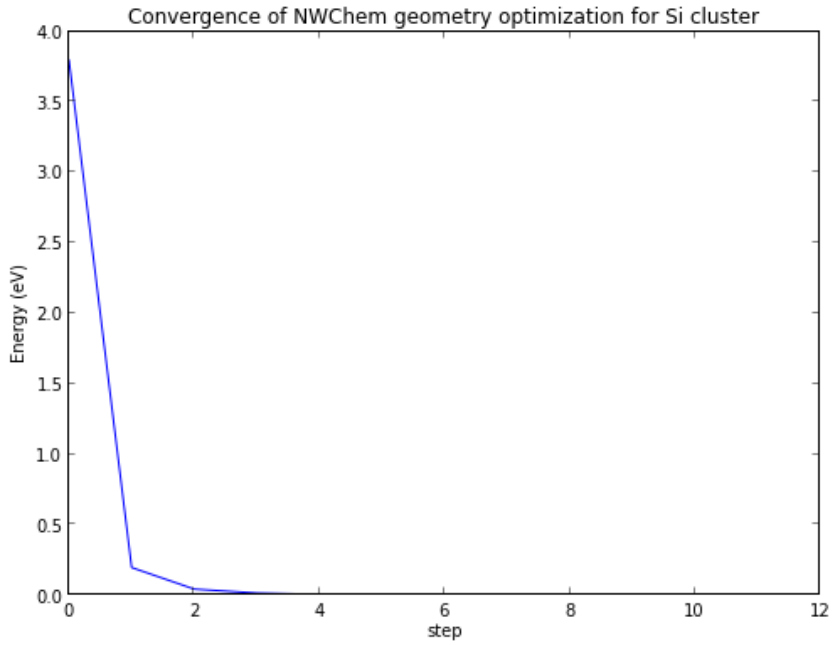
<matplotlib.text.Text at 0x111172690>



Les unités **Hartree** - que la plupart des logiciels de chimie utilisent par défaut - sont vraiment des unités stupides. On préférera travailler avec des kcal/mol ou des eV ou quelque unité vraiment utile. Alors, retraçons vite fait les données avec des eV au-dessus de l'énergie minimum, ce qui nous fournira un graphique plus d'à-propos :

In :
```
energies = data[:,0]
minE = min(energies)
energies_eV = 27.211*(energies-minE)
plot(energies_eV)
xlabel('step')
ylabel('Energy (eV)')
title('Convergence of NWChem geometry optimization for Si cluster')
```

<matplotlib.text.Text at 0x111196ed0>



Le résultat est fourni dans un format qui facilite l'analyse : 4 eV est un changement important - les liaisons chimiques sont à peu près de cet ordre de grandeur énergétique - et la plupart de la diminution d'énergie a été obtenue dès la première itération géométrique.

Nous avons remarqué précédemment que nous n'étions pas obligés de nous appuyer sur grep pour extraire les lignes intéressantes. Le module Python standard string dispose de nombreuses fonctions très utiles. Parmi celles-ci, la fonction startswith(). Par exemple :

In :
```
lines = """\
                 ----------------------------------------
                 |  WALL  |       0.45   |     443.61   |
                 ----------------------------------------

@ Step        Energy      Delta E   Gmax     Grms     Xrms     Xmax   Walltime
@ ---- ---------------- -------- -------- -------- -------- -------- --------
@    0   -6095.12544083  0.0D+00  0.03686  0.00936  0.00000  0.00000   1391.5
                                                      ok       ok



                                Z-matrix (autoz)
                                --------
""".splitlines()

for line in lines:
    if line.startswith('@'):
        print line
```

Out :
```
@ Step        Energy      Delta E   Gmax     Grms     Xrms     Xmax   Walltime
@ ---- ---------------- -------- -------- -------- -------- -------- --------
@    0   -6095.12544083  0.0D+00  0.03686  0.00936  0.00000  0.00000   1391.5
```

Et nous obtenons ainsi toutes les lignes commençant par le signe arobase (@).

La véritable valeur ajoutée - avec un langage comme Python - est qu'il devient facile d'ajouter des traitements pour analyser les données, ce qui signifie que vous vous consacrez plus à vos données qu'au codage du programme et que vous êtes plus susceptible de mettre en exergue des points importants dans vos analyses.

III - Aller plus loin avec le formatage et les traitements des chaînes de caractères

Les chaînes de caractères engendrent souvent des problèmes épineux dans la plupart des langages de programmation modernes et j'espère que les rubriques précédentes ont bien montré à quel point les techniques de manipulation de chaînes de caractères Python sont puissantes et polyvalentes.

Nous pouvons afficher des lignes en Python avec le mot-clé réservé print.

In :
```
print "I have 3 errands to run"
```

```
I have 3 errands to run
```

Dans la console interactive IPython, nous n'avons même pas besoin de la commande print, étant donné que la console affichera automatiquement toute expression autre qu'une initialisation de variable.

In :
```
"I have 3 errands to run"
```

Out :
```
'I have 3 errands to run'
```

La commande print convertit même certains arguments en chaînes de caractères pour nous :

In :
```
a,b,c = 1,2,3
print "The variables are ",a,b,c
```

```
The variables are  1 2 3
```

Si puissant que tout cela paraisse, vous avez malgré tout besoin de plus de latitude d'action sur les données que vous voulez afficher. Par exemple, qu'en est-il si nous voulons afficher une portion de données avec quatre décimales après la virgule ? Nous pouvons obtenir cela avec le formatage de chaînes de caractères.

Le formatage de chaînes de caractères Python partage quelques points communs avec la fameuse fonction C printf(). D'abord, on crée une chaîne de caractères contenant de drôles de *caractères de formatage* , puis nous appliquons une série de variables à cette chaîne de caractères dont les valeurs viendront remplacer ces caractères de formatage selon des **règles précises**.

Par exemple :

In :
```
print "Pi as a decimal = %d" % pi
print "Pi as a float = %f" % pi
print "Pi with 4 decimal places = %.4f" % pi
print "Pi with overall fixed length of 10 spaces, with 6 decimal places = %10.6f" % pi
print "Pi as in exponential format = %e" % pi
```

```
Pi as a decimal = 3
Pi as a float = 3.141593
Pi with 4 decimal places = 3.1416
Pi with overall fixed length of 10 spaces, with 6 decimal places =   3.141593
Pi as in exponential format = 3.141593e+00
```

Nous utilisons ici le signe pourcentage (%) de deux manières différentes. Tout d'abord, le caractère de formatage lui-même débute avec le signe (%) dans la chaîne de caractères de formatage. Les types de données sont encodés avec des lettres : %d ou %i pour les **entiers**, %f pour les nombres à virgule flottante (*float*), %e pour la **notation exponentielle** des nombres à virgule flottante *e.g.* 1.23e-12, %s pour une chaîne de caractères (*str*) attendue, etc. Un nombre entier qui suit immédiatement le signe (%) indique la largeur du champ (disposition du texte en colonnes, tabulation) en nombre d'espaces blanches. Si ce nombre commence par 0, les valeurs numériques à formater sont complétées par des zéros - exemple print "%03d" % 1 donne **'001'**. Les nombres à virgule flottante acceptent de plus un nombre précédé du signe point final (.) indiquant le nombre de décimales après la virgule - exemple print "%03.2f" % 1.456 qui donne **'001.46'**.

L'autre utilisation du signe (%) vient **après** la chaîne de formatage et relie une liste de variables aux différents caractères de formatage. Vous devez passer **exactement** autant de variables en paramètres dans cette liste (*tuple*) que la chaîne de caractères de formatage ne déclare de formats à afficher. Ainsi,

In :
```
print "The variables specified earlier are %d, %d, and %d" % (a,b,c)
```

```
The variables specified earlier are 1, 2, and 3
```

Il s'agit là d'un modèle de formatage qui répondra à la plupart de vos besoins. Pour plus d'information, consultez cette **documentation**.

Il est important de noter qu'il existe une autre façon de formater des chaînes de caractères, notamment avec la méthode **format()** et son **minilangage**, mais je préfère néanmoins la technique ci-dessus pour sa simplicité de mise en œuvre et pour ses similitudes avec la fonction C printf().

Dans les paragraphes précédents, nous avions abordé les chaînes de caractères multilignes. Nous pouvons aussi mettre des caractères de formatage dans des chaînes de caractères de ce type.

In :
```
form_letter = """\

          %s

Dear %s,

We regret to inform you that your product did not
ship today due to %s.

We hope to remedy this as soon as possible.

          From,
          Your Supplier
"""

print form_letter % ("July 1, 2013","Valued Customer Bob","alien attack")
```

```
          July 1, 2013

Dear Valued Customer Bob,

We regret to inform you that your product did not
ship today due to alien attack.

We hope to remedy this as soon as possible.

          From,
          Your Supplier
```

Le problème avec les grands blocs de texte comme celui-ci, c'est qu'il devient difficile de savoir quel caractère de formatage doit correspondre à quelle variable dans la liste d'arguments. Il existe une notation alternative pour remédier à ce problème : on donne un nom entre parenthèses à chaque caractère de formatage, puis on remplace la liste d'arguments par un dictionnaire Python (*dict* ou *{}*) d'arguments nommés. Voici ce que cela donne :

In :
```
form_letter = """\

          %(date)s

Dear %(customer)s,

We regret to inform you that your product did not
ship today due to %(lame_excuse)s.

We hope to remedy this as soon as possible.

          From,
          Your Supplier
"""

print form_letter % {"date"
 : "July 1, 2013","customer":"Valued Customer Bob","lame_excuse":"alien attack"}
```

```
          July 1, 2013

Dear Valued Customer Bob,

We regret to inform you that your product did not
```

```
ship today due to alien attack.

We hope to remedy this as soon as possible.

          From,
          Your Supplier
```

Avec cette notation par arguments nommés, vous diminuez le risque d'erreur, comme par exemple d'afficher malencontreusement « alien attack » à la place du nom de votre client.

En tant que scientifiques, vous n'êtes peut-être pas très sensibles aux arguments de mailings commerciaux. Mais il existe des fonctionnalités pour produire des traitements similaires dans le domaine scientifique, comme de scanner diverses structures pour trouver la configuration optimale pour quelque chose.

Par exemple, vous pourriez utiliser le modèle suivant pour vos fichiers d'entrée NWChem :

In :
```
nwchem_format = """
start %(jobname)s

title "%(thetitle)s"
charge %(charge)d

geometry units angstroms print xyz autosym
%(geometry)s
end

basis
  * library 6-31G**
end

dft
  xc %(dft_functional)s
  mult %(multiplicity)d
end

task dft %(jobtype)s
"""
```

Si vous voulez produire plusieurs sorties d'un coup, il est plus facile d'associer un petit script à ce modèle, quitte à compléter un peu les formatages :

In :
```
oxygen_xy_coords = [(0,0),(0,0.1),(0.1,0),(0.1,0.1)]
charge = 0
multiplicity = 1
dft_functional = "b3lyp"
jobtype = "optimize"

geometry_template = """\
  O    %f     %f      0.0
  H    0.0    1.0     0.0
  H    1.0    0.0     0.0"""

for i,xy in enumerate(oxygen_xy_coords):
    thetitle = "Water run #%d" % i
    jobname = "h2o-%d" % i
    geometry = geometry_template % xy
    print "---------"
    print nwchem_format % dict(thetitle=thetitle,charge=charge,jobname=jobname,jobtype=jobtype,

 geometry=geometry,dft_functional=dft_functional,multiplicity=multiplicity)
```

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

start h2o-0
```

```
title "Water run #0"
charge 0

geometry units angstroms print xyz autosym
  O    0.000000     0.000000      0.0
  H    0.0    1.0     0.0
  H    1.0    0.0     0.0
end

basis
  * library 6-31G**
end

dft
  xc b3lyp
  mult 1
end

task dft optimize

---------

start h2o-1

title "Water run #1"
charge 0

geometry units angstroms print xyz autosym
  O    0.000000     0.100000      0.0
  H    0.0    1.0     0.0
  H    1.0    0.0     0.0
end

basis
  * library 6-31G**
end

dft
  xc b3lyp
  mult 1
end

task dft optimize

---------

start h2o-2

title "Water run #2"
charge 0

geometry units angstroms print xyz autosym
  O    0.100000     0.000000      0.0
  H    0.0    1.0     0.0
  H    1.0    0.0     0.0
end

basis
  * library 6-31G**
end

dft
  xc b3lyp
  mult 1
end

task dft optimize

---------

start h2o-3
```

```
title "Water run #3"
charge 0

geometry units angstroms print xyz autosym
  O    0.100000     0.100000      0.0
  H    0.0    1.0     0.0
  H    1.0    0.0     0.0
end

basis
  * library 6-31G**
end

dft
  xc b3lyp
  mult 1
end

task dft optimize
```

C'est une bien mauvaise géométrie pour une molécule d'eau - et ce serait ridicule de tester autant d'optimisations de structures qui de toute façon convergeront vers la même géométrie unique - mais ce code vous donne une bonne idée sur la manière de produire plusieurs sorties de résultat en une seule fois.

Nous avons utilisé la fonction enumerate() pour boucler sur les indices et les éléments d'une séquence en même temps. Cette fonction est à peu près l'équivalent de :

In :
```
def my_enumerate(seq):
    l = []
    for i in range(len(seq)):
        l.append((i,seq[i]))
    return l
my_enumerate(oxygen_xy_coords)
```

Out :
```
[(0, (0, 0)), (1, (0, 0.1)), (2, (0.1, 0)), (3, (0.1, 0.1))]
```

bien qu'en réalité la fonction standard utilise des **générateurs** (voir plus bas) qui la dispense de créer de trop grandes listes - gourmandes en mémoire - et qui permettent d'accélérer notablement les calculs sur de très longues séquences.

IV - Arguments facultatifs

Vous vous souvenez sans doute de la fonction objet linspace() qui peut prendre deux arguments (début, fin) :

In :
```
linspace(0,1)
```

Out :
```
array([ 0.        ,  0.02040816,  0.04081633,  0.06122449,  0.08163265,
        0.10204082,  0.12244898,  0.14285714,  0.16326531,  0.18367347,
        0.20408163,  0.2244898 ,  0.24489796,  0.26530612,  0.28571429,
        0.30612245,  0.32653061,  0.34693878,  0.36734694,  0.3877551 ,
        0.40816327,  0.42857143,  0.44897959,  0.46938776,  0.48979592,
        0.51020408,  0.53061224,  0.55102041,  0.57142857,  0.59183673,
        0.6122449 ,  0.63265306,  0.65306122,  0.67346939,  0.69387755,
        0.71428571,  0.73469388,  0.75510204,  0.7755102 ,  0.79591837,
        0.81632653,  0.83673469,  0.85714286,  0.87755102,  0.89795918,
        0.91836735,  0.93877551,  0.95918367,  0.97959184,  1.        ])
```

et vous vous souvenez certainement aussi de la même fonction avec trois arguments (début, fin, nombre de points) :

In :
```
linspace(0,1,5)
```

Out :
```
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

Vous pouvez même rajouter encore un argument nommé pour exclure la borne supérieure :

In :
```
linspace(0,1,5,endpoint=False)
```

Out :
```
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

Jusqu'à présent, nous ne savons définir que des fonctions comportant un nombre fixe d'arguments. Voyons comment nous pourrions généraliser certains cas.

Si nous définissions nous-même une version simple de linspace(), nous pourrions écrire :

In :
```
def my_linspace(start,end):
    npoints = 50
    v = []
    d = (end-start)/float(npoints-1)
    for i in range(npoints):
        v.append(start + i*d)
    return v
my_linspace(0,1)
```

Out :
```
[0.0,
 0.02040816326530612,
 0.04081632653061224,
 0.061224489795918366,
 0.08163265306122448,
 0.1020408163265306,
 0.12244897959183673,
 0.14285714285714285,
 0.16326530612244897,
 0.18367346938775508,
 0.2040816326530612,
 0.22448979591836732,
 0.24489795918367346,
 0.26530612244897955,
 0.2857142857142857,
 0.3061224489795918,
 0.32653061224489793,
 0.3469387755102041,
 0.36734693877551017,
 0.3877551020408163,
 0.4081632653061224,
 0.42857142857142855,
 0.44897959183673464,
 0.4693877551020408,
 0.4897959183673469,
 0.5102040816326531,
 0.5306122448979591,
 0.5510204081632653,
 0.5714285714285714,
 0.5918367346938775,
 0.6122448979591836,
 0.6326530612244897,
 0.6530612244897959,
 0.673469387755102,
 0.6938775510204082,
 0.7142857142857142,
```

Out :
```
 0.7346938775510203,
 0.7551020408163265,
 0.7755102040816326,
 0.7959183673469387,
 0.8163265306122448,
 0.836734693877551,
 0.8571428571428571,
 0.8775510204081632,
 0.8979591836734693,
 0.9183673469387754,
 0.9387755102040816,
 0.9591836734693877,
 0.9795918367346939,
 0.9999999999999999]
```

Voici comment ajouter un argument facultatif avec sa valeur par défaut dans la liste des arguments de la fonction :

In :
```
def my_linspace(start,end,npoints = 50):
    v = []
    d = (end-start)/float(npoints-1)
    for i in range(npoints):
        v.append(start + i*d)
    return v
```

Nous voyons qu'ici l'argument **npoints** prendra la valeur par défaut **50,** si on ne le déclare pas durant un appel de la fonction :

In :
```
my_linspace(0,1)
```

Out :
```
[0.0,
 0.02040816326530612,
 0.04081632653061224,
 0.061224489795918366,
 0.08163265306122448,
 0.1020408163265306,
 0.12244897959183673,
 0.14285714285714285,
 0.16326530612244897,
 0.18367346938775508,
 0.2040816326530612,
 0.22448979591836732,
 0.24489795918367346,
 0.26530612244897955,
 0.2857142857142857,
 0.3061224489795918,
 0.32653061224489793,
 0.3469387755102041,
 0.36734693877551017,
 0.3877551020408163,
 0.4081632653061224,
 0.42857142857142855,
 0.44897959183673464,
 0.4693877551020408,
 0.4897959183673469,
 0.5102040816326531,
 0.5306122448979591,
 0.5510204081632653,
 0.5714285714285714,
 0.5918367346938775,
 0.6122448979591836,
 0.6326530612244897,
 0.6530612244897959,
 0.673469387755102,
 0.6938775510204082,
```

Out :
```
  0.7142857142857142,
  0.7346938775510203,
  0.7551020408163265,
  0.7755102040816326,
  0.7959183673469387,
  0.8163265306122448,
  0.836734693877551,
  0.8571428571428571,
  0.8775510204081632,
  0.8979591836734693,
  0.9183673469387754,
  0.9387755102040816,
  0.9591836734693877,
  0.9795918367346939,
  0.9999999999999999]
```

Mais grâce au fait que nous l'avons déclaré en tant qu'argument, nous pouvons désormais changer sa valeur directement dans un appel de la fonction :

In :
```
my_linspace(0,1,5)
```

Out :
```
[0.0, 0.25, 0.5, 0.75, 1.0]
```

Nous pouvons même ajouter des arguments nommés dans la définition de la fonction grâce à la notation ****nom_variable** comme suit :

In :
```
def my_linspace(start,end,npoints=50,**kwargs):
    endpoint = kwargs.get('endpoint',True)
    v = []
    if endpoint:
        d = (end-start)/float(npoints-1)
    else:
        d = (end-start)/float(npoints)
    for i in range(npoints):
        v.append(start + i*d)
    return v
my_linspace(0,1,5,endpoint=False)
```

Out :
```
[0.0, 0.2, 0.4, 0.6000000000000001, 0.8]
```

La notation ****nom_variable** placée dans la définition d'une fonction permet de regrouper des paramètres que l'on initialise nommément lors de l'appel de la fonction - exemple : **ma_fonction(nom="CAVERE", prenom="Nicole", age=-20000, etc.)** - à l'intérieur d'un dictionnaire Python (*dict* ou *{}*), dans notre exemple de code, il s'agit du dictionnaire **kwargs**. Comme les paramètres sont regroupés dans un seul dictionnaire d'arguments nommés, vous pouvez y accéder comme n'importe quel élément de dictionnaire Python standard. Par précaution, on utilise la méthode **https://docs.python.org/2/library/stdtypes.html#dict.get(key, default_value)** qui permet de récupérer la valeur de l'argument nommé (**key**) ou une valeur par défaut (**default_value**), si l'argument nommé a été omis lors de l'appel de la fonction. Cette technique demande un peu de gymnastique au début, mais elle est courante en programmation Python et vous la rencontrerez certainement à plusieurs reprises.

*NdT : la terminologie pour la notation ****nom_variable** est « liste variable d'arguments nommés ».*

Il existe aussi un pendant « liste variable d'arguments » non nommés (anonymes) qui s'écrit ***nom_variable** (avec une seule étoile au lieu de deux) et qui place toute liste de valeurs passées lors d'un appel de fonction dans une liste Python standard (*list* ou *[]*). Pensez par exemple à la fonction range() : elle peut prendre un seul argument - range(n) - deux arguments - range(start, stop) - ou même trois arguments - range(start, stop, step). Comment pourrions-nous définir une telle fonction ?

In :
```
def my_range(*args):
    start, stop, step = 0, 0, 1
    l = len(args)
    if l == 1:
        stop = args[0]
    elif l == 2:
        start, stop = args
    elif l == 3:
        start, stop, step = args
    else:
        raise Exception("Unable to parse arguments")
    v = []

    while start < stop:
        v.append(start)
        start += step
    return v
```

Vous aurez remarqué que nous avons utilisé ici le mot-clé raise qui permet de lever une *exception* (erreur prise en charge par l'interpréteur Python lui-même) et d'afficher un message personnalisé pour détailler l'erreur en question. Par exemple :

In :
```
my_range()
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-290-0e8004dab150> in <module>()
----> 1 my_range()

<ipython-input-289-c34e09da2551> in my_range(*args)
      9         start,end,step = args
     10     else:
---> 11         raise Exception("Unable to parse arguments")
     12     v = []
     13     value = start

Exception: Unable to parse arguments
```

V - Listes en compréhension et générateurs

Les listes en compréhension sont un moyen simple et puissant de produire des listes Python (*list* ou *[]*). Leur notation fait penser à une définition de liste contenant des instructions bien précises. Par exemple :

In :
```
evens1 = [2*i for i in range(10)]
print evens1
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

On peut aussi rajouter un test à la volée pour chaque élément parcouru par l'itération :

In :
```
odds = [i for i in range(20) if i%2==1]
odds
```

Out :
```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Ici, l'expression i % 2 correspond à **i modulo 2** ce qui produit bien la valeur booléenne True pour le test if i % 2 == 1 lorsque **i est impair**, et qui explique que seules les valeurs impaires sont retenues pour la fabrication de la liste.

Les **itérateurs** sont un autre moyen de parcourir des séquences. Supposons que nous ayons deux boucles imbriquées l'une dans l'autre :

```
for i in range(1000000):
    for j in range(1000000):
```

Dans la première boucle, nous créons une liste d'un million (1 000 000) d'entiers juste pour les parcourir un à un, l'un après l'autre. Nous n'avons pas besoin de fonctionnalités spécifiques aux listes Python, telles que l'échantillonnage ou l'accès aléatoire, nous avons juste besoin de parcourir un nombre à la fois. Mais nous avons accaparé la mémoire avec un million (1 000 000) d'éléments pour faire cela et ce n'est pas rien.

Les **itérateurs** permettent de contourner ce problème. Par exemple, la fonction xrange() fait la même chose que la fonction range() mais en se servant d'un **itérateur**. Elle crée simplement un compteur qui avance en pointant sur une séquence virtuelle, qui est en réalité calculée au fur et à mesure des itérations que la boucle for … in … effectue, à la demande. Nous pouvons donc écrire :

```
for i in xrange(1000000):
    for j in xrange(1000000):
```

Quand bien même nous avons seulement remplacé range() par xrange(), nous avons en réalité substantiellement accéléré le code, ne serait-ce que parce que nous ne générons plus deux listes à un million d'éléments chacune.

Nous pouvons créer nos propres **itérateurs** en utilisant le mot-clé réservé yield :

In :
```
def evens_below(n):
    for i in xrange(n):
        if i%2 == 0:
            yield i
    return

for i in evens_below(9):
    print i
```

Out :
```
0
2
4
6
8
```

Ensuite, rien ne nous empêche d'en tirer une liste grâce à la fonction objet list() :

In :
```
list(evens_below(9))
```

Out :
```
[0, 2, 4, 6, 8]
```

Il existe une syntaxe particulière appelée **expression génératrice** qui ressemble peu ou prou à une liste en compréhension (remarquez les parenthèses à la place des crochets) :

In :
```
evens_gen = (i for i in xrange(9) if i%2==0)
for i in evens_gen:
    print i
```

```
0
2
4
```

```
6
8
```

VI - Fonctions de fonctions (fermetures)

Une fonction dite « fermeture » est une fonction qui retourne en résultat une autre fonction. Cette fonction porte le joli nom de **clôture** *lexicale*, qui vous fait paraître drôlement intelligent(e) en compagnie de vos amis informaticiens, lorsque vous le prononcez. Cet aspect quelque peu ésotérique mis à part, les fermetures peuvent rendre de grands services.

En supposant que vous vouliez une fonction gaussienne centrée sur 0,5, avec une hauteur de 99 et une largeur de 1,0. Vous pourriez écrire une fonction générique :

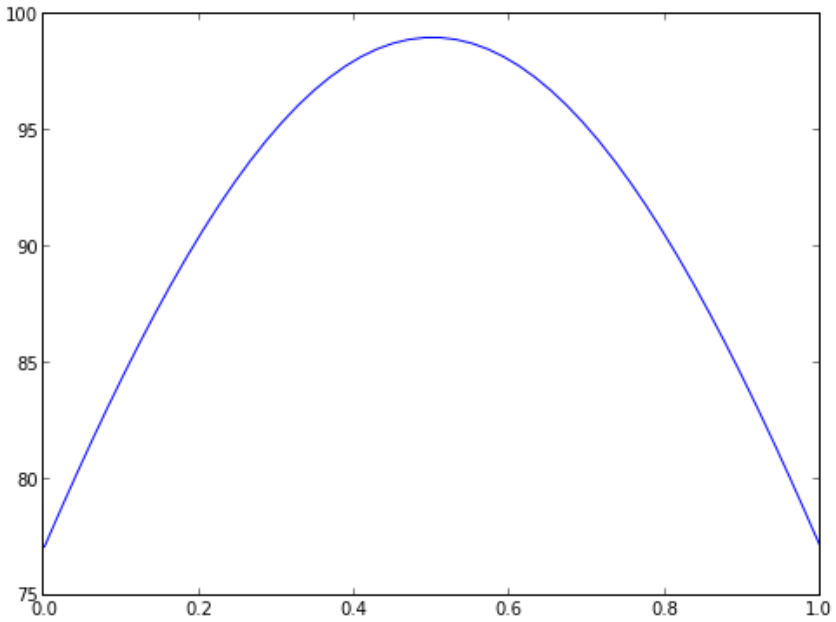
In :
```
def gauss(x,A,a,x0):
    return A*exp(-a*(x-x0)**2)
```

Mais qu'en serait-il si vous aviez besoin d'une fonction avec un seul argument, genre **f(x)** plutôt que **f(x, y, z, …)** ? Voici comment le faire avec une fermeture :

In :
```
def gauss_maker(A,a,x0):
    def f(x):
        return A*exp(-a*(x-x0)**2)
    return f
```

In :
```
x = linspace(0,1)
g = gauss_maker(99.0,1.0,0.5)
plot(x,g(x))
```

[<matplotlib.lines.Line2D at 0x1114ca090>]



Dans Python, tout est objet, y compris les fonctions. Cela implique, entre autres, qu'une fonction peut être retournée en résultat d'une autre fonction comme n'importe quel objet. On peut même passer une fonction en argument d'une autre fonction (fonction de rappel dite **callback**), mais là n'est pas le sujet de notre discussion. Dans notre exemple de fermeture `gauss_maker()` , la fonction `g` conserve en mémoire les valeurs des paramètres (A, a, x0) qui ont servi à

l'initialiser - *i.e.* dans `g = gauss_maker(99.0,1.0,0.5)` - étant donné que ces valeurs sont stockées en tant que locales à la fonction `gauss_maker()` (c'est d'ailleurs à ce principe que se réfère la notion de *clôture lexicale*).

Les fermetures constituent une part importante des **patrons de conception**, un ensemble de règles à suivre pour produire des logiciels de haute qualité, portables, lisibles et stables. Tout cela va bien au-delà des objectifs du présent document, mais je pensais que c'était utile de le mentionner pour les personnes qui seraient intéressées par la conception de logiciels.

VII - La sérialisation : conserver pour plus tard

Le principe de la **sérialisation** consiste à exporter des données - et occasionnellement des fonctions - vers une base de données ou vers un fichier, dans le but de s'en servir à nouveau plus tard. Dans les tous premiers jours des langages de programmation, la sérialisation se faisait dans des fichiers texte. Python est excellent en matière de traitement de textes et vous en savez probablement suffisamment pour pouvoir vous en servir.

Lorsque l'accès à de vastes champs de données devint crucial, les informaticiens développèrent des logiciels de gestion de bases de données autour du standard SQL (**Structured Query Language**). Je ne vais pas traiter de SQL ici, mais si cela vous intéresse, je vous recommande le module Python standard **sqlite3**.

Avec la montée en puissance du partage de données est apparu le XML (**eXtensible Markup Language**). XML formate les données de telle sorte qu'il est facile d'écrire des analyseurs pour ce langage, en simplifiant grandement les ambiguïtés qui pourraient apparaître durant un traitement. Encore une fois, je ne vais pas traiter de XML ici, mais si le sujet vous intéresse, jetez un œil dans le module Python standard **xml.etree.ElementTree** (analyseurs et arborescences d'éléments XML).

Python dispose d'un format générique de sérialisation appelé **pickle** qui peut transformer n'importe quel objet Python - y compris des fonctions et des classes - en une structure qui peut être écrite dans un fichier puis relue plus tard. Là encore, je ne vais pas m'étendre sur le sujet, étant donné que je ne l'utilise que très rarement. Pour en savoir plus, consultez la documentation Python standard sur le module **pickle** et sa variante optimisée **cPickle**.

Je vais plutôt vous parler d'un format relativement récent, le JSON (**JavaScript Object Notation**) qui est devenu vraiment populaire ces dernières années. Vous disposez d'un module Python standard **json** pour encoder et décoder dans ce format de données. La raison principale pour laquelle j'aime tant JSON, c'est que cela ressemble un peu à Python, ainsi, contrairement aux autres options envisagées plus haut, vous pouvez jeter un œil dans vos données, les modifier, les utiliser avec d'autres programmes, etc.

Petit exemple :

In :
```
# les données sont au format JSON
json_data = """\
{
    "a": [1,2,3],
    "b": [4,5,6],
    "greeting" : "Hello"
}"""
import json
json.loads(json_data)
```

Out :
```
{u'a': [1, 2, 3], u'b': [4, 5, 6], u'greeting': u'Hello'}
```

Ne faites pas trop attention aux petits **u** en début de chaînes de caractères, il s'agit d'une notation Python pour indiquer que les chaînes de caractères sont UNICODE (**documentation**). Vos données sont écrites dans ce qui ressemble à un objet dictionnaire Python (*dict* ou *{}*) et en une seule ligne de code, vous pouvez transférer ces données directement dans un vrai dictionnaire Python pour un usage ultérieur.

De la même façon, vous pouvez - toujours en une seule ligne de code - mettre tout un tas de variables dans un dictionnaire Python puis écrire le résultat dans un fichier :

In :
```
json.dumps({"a":[1,2,3],"b":[9,10,11],"greeting":"Hola"})
```

Out :
```
'{"a": [1, 2, 3], "b": [9, 10, 11], "greeting": "Hola"}'
```

VIII - Programmation fonctionnelle

La **programmation fonctionnelle** est un vaste sujet. En gros, l'idée c'est d'avoir une série de fonctions qui génèrent chacune une nouvelle structure de données à partir d'une entrée, sans jamais modifier la structure de l'entrée elle-même. En ne modifiant pas la structure de l'entrée (on dit aussi *ne pas avoir d'***effets de bord**), on peut garantir l'indépendance des processus, ce qui peut faciliter le **parallélisme** et la précision des programmes. Il existe un document **Python Functional Programming HOWTO** qui traite du sujet plus en profondeur. Je voulais juste exposer ici les grandes lignes.

Vous disposez d'un module Python standard **operator** qui contient des versions fonctionnelles de la plupart des opérateurs natifs du langage. Par exemple :

In :
```
from operator import add, mul
add(1,2)
```

Out :
```
3
```

In :
```
mul(3,4)
```

Out :
```
12
```

Ce sont des blocs de construction utiles pour la programmation fonctionnelle.

Le mot-clé réservé `lambda` permet de définir des **fonctions anonymes**, qui sont tout simplement des fonctions qui ne sont pas définies avec le mot-clé standard `def` suivi d'un nom de fonction. Par exemple, une fonction qui retourne le double de la valeur d'entrée s'écrit :

In :
```
def doubler(x): return 2*x
doubler(17)
```

Out :
```
34
```

Ce qui, pour une fonction anonyme, s'écrit :

In :
```
lambda x: 2*x
```

Out :
```
<function __main__.<lambda>>
```

Il suffit ensuite de l'assigner à une variable :

In :
```
another_doubler = lambda x: 2*x
another_doubler(19)
```

Out :
```
38
```

L'utilisation de lambda est particulièrement pratique - comme nous le verrons un peu plus bas - lorsque l'on souhaite définir une fonction - à la volée - en tant qu'argument d'une autre fonction.

La fonction objet map() permet d'appliquer une fonction à chaque élément d'une séquence donnée, puis de retourner le résultat final dans une liste Python (*list* ou *[]*) :

In :
```
map(float, (1,2,3,4,5))
```

Out :
```
[1.0, 2.0, 3.0, 4.0, 5.0]
```

La fonction reduce() permet de réduire progressivement une séquence donnée en lui appliquant une fonction à **deux** arguments, le premier argument étant le résultat de l'application de la fonction à l'étape précédente et le second argument le prochain élément à considérer *e.g.* reduce(lambda x,y: x+y, (1,2,3,4,5)) calcule d'abord (1+2), puis ((3)+3), puis ((6)+4), puis ((10)+5) pour obtenir finalement 15 (*cf.* **explication détaillée**). La fonction sum() qui calcule la somme des éléments d'une séquence donnée est une fonction de réduction similaire à reduce(lambda x,y: x+y, sequence) :

In :
```
sum([1,2,3,4,5])
```

Out :
```
15
```

Nous pouvons utiliser reduce() pour définir une fonction prod() qui multiplie entre eux les éléments d'une séquence donnée :

In :
```
def prod(l): return reduce(mul,l)
prod([1,2,3,4,5])
```

Out :
```
120
```

IX - Programmation Orientée Objet (POO)

Nous avons vu de nombreux cas d'**objets** dans Python. Par exemple, nous créons un **objet** chaîne de caractères avec des guillemets :

In :
```
mystring = "Hi there"
```

et nous avons un tas de **méthodes** que nous pouvons utiliser avec cet objet :

In :
```
mystring.split()
```

Out :
```
['Hi', 'there']
```

In :
```
mystring.startswith('Hi')
```

Out :
```
True
```

In :
```
len(mystring)
```

Out :
```
8
```

La programmation orientée objet (**POO**) vous procure les outils nécessaires pour créer vous-mêmes vos propres objets et vos propres méthodes. C'est utile chaque fois que vous voulez conserver des données (comme les caractères dans la chaîne de caractères) étroitement liées aux fonctionnalités qui agissent spécifiquement sur elles *e.g.* str.split(), str.startswith(), str.replace(), etc.

En exemple d'application, nous allons regrouper certaines fonctions que nous avons créées précédemment, notamment pour redéfinir les fonctions de valeurs propres de l'oscillateur harmonique unidimensionnel avec des potentiels arbitraires (voir plus haut) - comme ça nous pourrons passer en argument de fonction une fonction définissant ce potentiel - quelques autres spécifications supplémentaires, puis produire quelque chose qui puisse tracer les orbitales, voire autre chose, si besoin est.

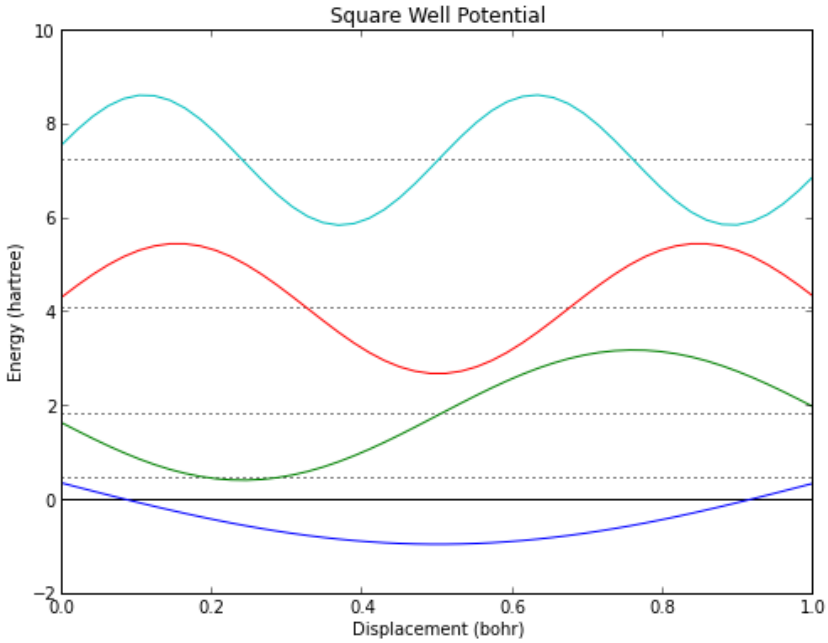
In :
```
class Schrod1d:
    """\
    Schrod1d: Solveur pour l'équation unidimensionnelle de Schrödinger.
    """
    def __init__(self,V,start=0,end=1,npts=50,**kwargs):
        m = kwargs.get('m',1.0)
        self.x = linspace(start,end,npts)
        self.Vx = V(self.x)
        self.H = (-0.5/m)*self.laplacian() + diag(self.Vx)
        return

    def plot(self,*args,**kwargs):
        titlestring = kwargs.get('titlestring',"Eigenfunctions of the 1d Potential")
        xstring = kwargs.get('xstring',"Displacement (bohr)")
        ystring = kwargs.get('ystring',"Energy (hartree)")
        if not args:
            args = [3]
        x = self.x
        E,U = eigh(self.H)
        h = x[1]-x[0]

        # on trace le potentiel
        plot(x,self.Vx,color='k')

        for i in range(*args):
            # on trace le niveau d'énergie pour chacune des premières solutions :
            axhline(y=E[i],color='k',ls=":")
            # de même que les valeurs propres, décalées par le niveau d'énergie
            # de sorte qu'elles ne s'empilent pas les unes sur les autres :
            plot(x,U[:,i]/sqrt(h)+E[i])
        title(titlestring)
        xlabel(xstring)
        ylabel(ystring)
        return

    def laplacian(self):
        x = self.x
        h = x[1]-x[0] # on suppose les points répartis uniformément
```

In :
```
        n = len(x)
        M = -2*identity(n,'d')
        for i in range(1,n):
            M[i,i-1] = M[i-1,i] = 1
        return M/h**2
```

La méthode __init__() est ce que l'on appelle communément un **constructeur** de la classe, c'est-à-dire la méthode qui va prendre en charge toutes les opérations d'initialisation lorsque l'objet sera créé. L'argument self est l'objet lui-même, cet argument est passé automatiquement par le langage Python à toutes les méthodes de la classe. Le seul argument obligatoire est la fonction définissant le potentiel QM. Nous pouvons aussi spécifier des arguments supplémentaires, qui définiront la grille numérique que nous allons utiliser pour nos calculs.

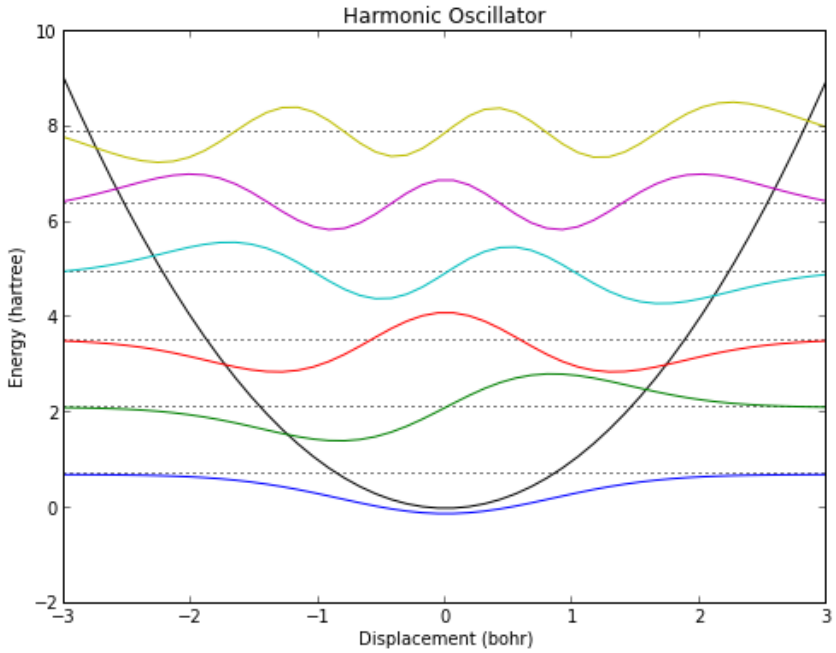
Par exemple, pour faire un puits de potentiel carré avec barrière infinie, nous avons une fonction qui vaut 0 partout. Inutile de préciser les bornes, étant donné que nous ne définirons le potentiel qu'à l'intérieur du puits, ce qui implique qu'il ne peut être défini ailleurs.

In :
```
square_well = Schrod1d(lambda x: 0*x,m=10)
square_well.plot(4,titlestring="Square Well Potential")
```



Nous pouvons redéfinir le potentiel de l'oscillateur harmonique de façon similaire.

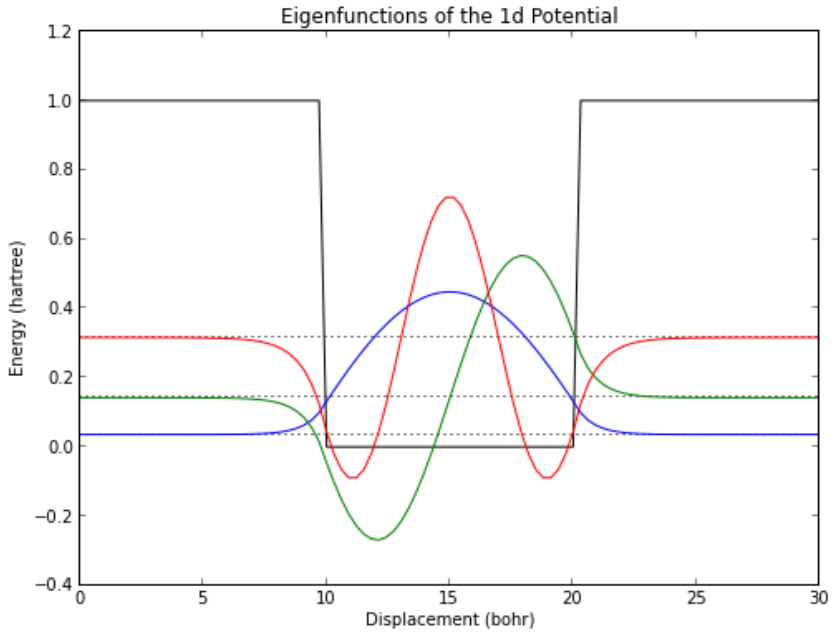
In :
```
ho = Schrod1d(lambda x: x**2,start=-3,end=3)
ho.plot(6,titlestring="Harmonic Oscillator")
```



Définissons à présent un puits de potentiel à barrière finie :

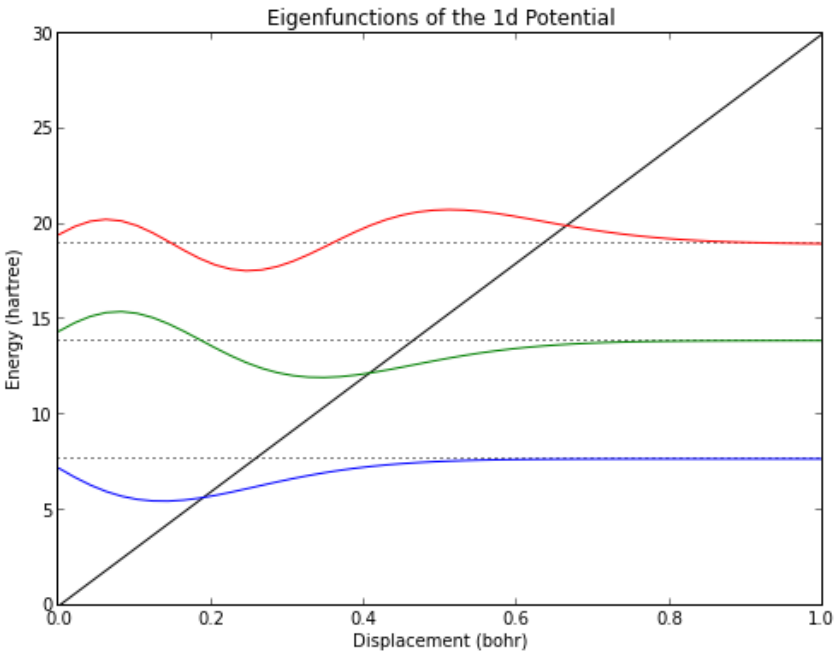
In :

```
def finite_well(x,V_left=1,V_well=0,V_right=1,d_left=10,d_well=10,d_right=10):
    V = zeros(x.size,'d')
    for i in range(x.size):
        if x[i] < d_left:
            V[i] = V_left
        elif x[i] > (d_left+d_well):
            V[i] = V_right
        else:
            V[i] = V_well
    return V

fw = Schrod1d(finite_well,start=0,end=30,npts=100)
fw.plot()
```



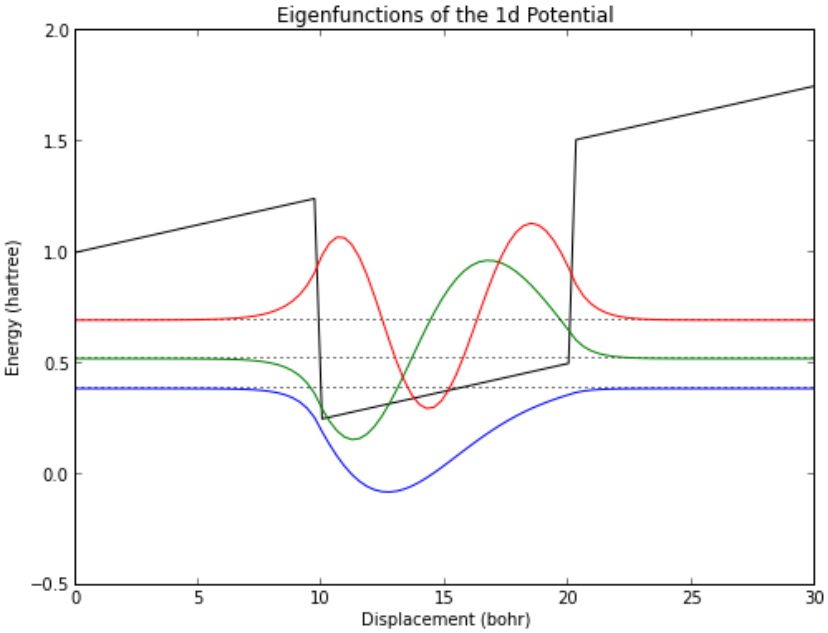
Un puits triangulaire :

In :
```
def triangular(x,F=30): return F*x

tw = Schrod1d(triangular,m=10)
tw.plot()
```



Nous pouvons aussi combiner les deux, pour obtenir quelque chose qui ressemble à un puits quantique de semi-conducteur avec porte haute :

In :
```
def tri_finite(x): return finite_well(x)+triangular(x,F=0.025)

tfw = Schrod1d(tri_finite,start=0,end=30,npts=100)
tfw.plot()
```



La programmation orientée objet, c'est tout un art, tout une philosophie. Comme je m'attache à ne présenter que les bases dans ce document, je n'irai pas plus loin, mais l'Internet regorge de ressources en tout genre sur le sujet. Pour ma part, je ne puis que trop vous recommander la lecture du fabuleux ouvrage **Design Patterns**.

À suivre (en cours de traduction) :

- *Partie 4 - Optimiser le code.*

X - Remerciements

Un grand merci à Alex et Tess pour tout !

Remerciements chaleureux à Barbara Muller et Tom Tarman pour leurs précieuses suggestions.

Sandia est un laboratoire pluridisciplinaire géré par la Sandia Corporation®, une filiale de la Lockheed Martin Company®, pour le compte des États-Unis d'Amérique, département de l'Énergie, administration de la Sûreté Nucléaire, sous contrat n° DE-AC04-94AL85000.

XI - Remerciements Developpez

Nous remercions Rick Muller qui nous a aimablement autorisé à traduire son cours **A Crash Course in Python for Scientists**.

Nos remerciements à Raphaël SEBAN (**tarball69**) pour la traduction et à Fabien (**f-leb**) pour la mise au gabarit.

Nous remercions également Jacques THÉRY (**jacques-jean**) pour sa relecture orthographique.